2

Productivity Engineering in the UNIX[1] Environment

Replicated Distributed Programs

Technical Report

S. L. Graham
Principal Investigator

(415) 642-2059

Contract No. N00039-84-C-0089

~~August 7, 1984 - August 6, 1987~~
april 1985

Arpa Order No. 4871

[1]UNIX is a trademark of AT&T Bell Laboratories

Replicated Distributed Programs

Eric Charles Cooper

April 1985

This report reproduces a dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate Division of the University of California, Berkeley.

This work was partially sponsored by a National Science Foundation Graduate Fellowship and by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4031, monitored by the Naval Electronics Systems Command under contract No. N00039-C-0235. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the U.S. Government.

Replicated Distributed Programs

Abstract

This dissertation presents a new software architecture for fault-tolerant distributed programs. This new architecture allows replication to be added transparently and flexibly to existing programs. Tuning the availability of a replicated program becomes a programming-in-the-large problem that a programmer need address only after the individual modules have been written and verified.

The increasing reliance that people place on computer systems makes it essential that those systems remain available. The low cost of computer hardware and the high cost of computer software make replicated distributed programs an attractive solution to the problem of providing fault-tolerant operation.

A *troupe* is a set of replicas of a module, executing on machines that have independent failure modes. Troupes are the building blocks of replicated distributed programs and the key to achieving high availability. Individual members of a troupe do not communicate among themselves, and are unaware of one another's existence; this property is what distinguishes troupes from other software architectures for fault tolerance.

Replicated procedure call is introduced to handle the many-to-many pattern of communication between troupes. Replicated procedure call is an elegant and powerful way of expressing many distributed algorithms. The semantics of replicated procedure call can be summarized as exactly-once execution at all replicas.

An implementation of troupes and replicated procedure call is described. Experiments were conducted to measure the performance of this implementation; an analysis of the results of these experiments is presented.

The problem of concurrency control for troupes is examined, and algorithms for replicated atomic transactions are presented as a solution. Binding and reconfiguration mechanisms for replicated distributed programs are described, and the problem of when to replace failed troupe members is analyzed.

Several issues relating to programming languages and environments for reliable distributed applications are discussed. Integration of the replication mechanisms into current programming languages is accomplished by means of *stub compilers*. Four stub compilers are examined, and some lessons learned from them are presented. A language for specifying troupe configurations is described, and the design of a configuration manager, a programming-in-the-large tool for configuring replicated distributed programs, is presented.

See p1

-A-

To my family:
my parents, Herbert and Joan Cooper;
my brother, Paul Cooper;
and my wife, Naomi Siegel.

Contents

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank all of the people who have helped me in one way or another while I was writing this dissertation. I can only acknowledge a few of them here:

> my advisor, Robert Fabry, for providing excellent support, encouragement, and advice over the past five years, and for creating CSRG, the most exciting research group at Berkeley;
>
> Domenico Ferrari, for somehow finding time to give me useful comments and advice while running the Computer Science Division and the PROGRES group;
>
> Susan Graham and Leo Harrington, the other members of my thesis committee;
>
> Andrew Birrell, my sponsor at DEC Systems Research Center;
>
> my friends Earl Cohen and Daniel Halbert, for setting good examples;
>
> and above all, my wife, Naomi Siegel, for her love and support.

This work was supported by the Defense Advanced Research Projects Agency, a National Science Foundation Graduate Fellowship, Xerox Corporation, and Digital Equipment Corporation.

Chapter 1

Introduction

1.1 Statement of Problem

As society relies more and more on services provided by computer systems, the penalties associated with even temporary loss of those services become increasingly severe. Today, the immediate effects of computer system failures on humans can range from temporary frustration to financial loss to injury or death.

This dissertation addresses the problem of constructing highly available software systems. (The adjectives *highly available*, *fault-tolerant*, and *nonstop* will be used synonymously to describe a system that continues to operate despite failures of some of its components.) The goal is to construct programs that automatically tolerate crashes of the underlying hardware. The problems posed by incorrect software or by hardware failures other than crashes are only addressed briefly.

The key to tolerating component failures is replication; this approach was proposed by von Neumann thirty years ago [106]. The idea is to replicate each component to such a degree that the probability of all replicas failing becomes acceptably small. The advent of inexpensive distributed computing systems (consisting of computers connected together by a network) makes replication an attractive and practical means of tolerating hardware crashes.

Computers and local-area networks have become economical, but distributed software remains expensive. The problem is that programmers must already cope with the complexity of their application areas; distributed hardware, if made visible at the application level, presents further complications. Mechanisms for constructing distributed programs must therefore provide *transparency*: the fact that several machines are involved must be hidden from the programmer. Transparency is an important theme in this dissertation.

A good implementation of *remote procedure call* provides transparency when all machines are working correctly, but cannot hide failures from the programmer. Some of the machines executing a distributed program may crash while others continue to function; this is called a *partial failure*. In contrast, a conventional single-machine program has an all-or-nothing failure mode. Partial failures violate transparency and must be masked if programmers are to be freed from worrying about the details of crash recovery algorithms for their applications. The key to masking partial failures is replication.

The replication used to mask partial failures must itself be hidden from the programmer, or else replication will add more complexity than it hides. The mechanisms presented in this dissertation provide *replication transparency* because they are invisible at the programming-in-the-small level.

Current programming methodology stresses the importance of modularity as a means of coping with complexity. It is important to realize that different modules require different guarantees of availability: some modules are reliability bottlenecks just as some modules are performance bottlenecks.

The performance of a program can be tuned by first determining which modules are performance bottlenecks, and then optimizing those modules. By analogy, one can envision a new methodology for constructing highly available programs in which replication is also controlled on a module-by-module basis. Tuning the overall availability of a program becomes a programming-in-the-large problem that a programmer need address only after the individual modules have been written and verified. This dissertation provides the tools to support such a methodology.

The ability to vary replication on a per-module basis is desirable because it allows software systems to adapt gracefully to changing characteristics of the underlying hardware. Even if perfectly reliable hardware were possible, there would still be periods during which hardware would be unavailable: scheduled down-time for preventive maintenance or reconfiguration, for example. The mechanisms described in this dissertation permit software systems to be reconfigured, while they are executing, so that their services remain available during such periods.

The reliability of the underlying hardware of a system may degrade or improve. A computer might crash more frequently than it used to; another computer might be replaced by a more reliable model. In these cases, it should be possible to adjust the amount of fault tolerance provided in software accordingly. This can be accomplished with the mechanisms developed in this dissertation.

Incorporating replication on a per-module basis is more flexible than previous approaches, such as providing fault tolerance in hardware or writing it into the application software. The first method is too expensive because it uses reliable hardware everywhere, not just for critical modules. The second approach burdens the programmer with the complexity of a non-transparent mechanism.

The fundamental mechanisms presented in this dissertation are:

- *troupes*, or replicated modules, and
- *replicated procedure call*, a generalization of remote procedure call for many-to-many communication between troupes.

The following important property is what distinguishes troupes and replicated procedure call from previous software architectures for fault tolerance: individual members of a troupe do not communicate among themselves, and are unaware of one another's existence. This property is also what gives these mechanisms their flexibility and power: since each troupe member behaves as if it had no replicas, the degree of replication of a troupe can be varied dynamically, with no recompilation or relinking.

1.2 Motivation

Remote procedure call shows the power of transparency: distributed programs become no more difficult to write than conventional single-machine programs. This research began with the idea of coupling remote procedure call to software replication, in the hope of making fault-tolerant distributed programs similarly easy to construct.

The wealth of computing power available in the research internet at Berkeley, combined with the fragility of early distributed programs, emphasized the need for a set of operating system and programming language tools for constructing reliable distributed applications.

Previous papers presented the author's initial ideas about replicated procedure calls [18] and a description of the Circus system [19].

1.3 Plan of Dissertation

The approach taken in this research is to combine remote procedure call with replication on a per-module basis. This basic idea manifests itself in three intertwined aspects of distributed programs: reliability, communication, and synchronization. Troupes, or replicated modules, are the basis for reliability, replicated procedure calls provide communication, and replicated atomic transactions are used for synchronization.

Chapter 2 reviews the necessary background and related work in the fields of reliability, communication, and synchronization.

Chapter 3 presents a model of program construction and execution and describes how its abstractions can be implemented transparently in a distributed system. Troupes are introduced as the building blocks of replicated distributed programs, and replication transparency is discussed.

Chapter 4 generalizes the notion of remote procedure call to the case of transfer of control between troupes, resulting in replicated procedure calls. The semantics of replicated procedure calls are defined, and algorithms are presented that implement these semantics.

This dissertation uses replicated procedure call primarily as a means of adding replication to programs transparently, while preserving conventional single-machine semantics. But replicated procedure call is also important in its own right: it is sufficiently general to express many distributed algorithms elegantly, and can be implemented efficiently on local-area networks.

The protocols and algorithms used in the Circus implementation are described. Measurements of the performance of Circus replicated procedure calls are discussed, and a theoretical analysis of a more efficient implementation is presented.

The problem of synchronizing concurrent access to shared data arises in unreplicated programs; additional mechanisms are required in the replicated case. Chapter 5 presents the necessary extensions to atomic transactions for use in replicated distributed programs.

Chapter 6 examines the problem of binding together programs constructed from troupes, and presents algorithms for dynamically reconfiguring such programs. The cache invalidation problem arises when clients amortize the cost of interactions with a binding agent by retaining old results. In the case of binding for replicated distributed programs, the cache invalidation problem is shown to be more complicated and more critical than for unreplicated programs.

Mechanisms are described that allow crashed components of replicated distributed programs to be replaced; this is a special case of reconfiguration. In addition, a probabilistic model of troupe availability is used to analyze *when* defunct troupe members should be replaced. The availability of a troupe is derived from the lifetime and replacement time of individual troupe members and the degree of replication.

Chapter 7 discusses the issues that arise when integrating these replicated mechanisms into programming languages. The approach here is based on stub compilers; four remote and replicated procedure call implementations are surveyed, and a section is devoted to the lessons about programming languages and stub compilers that were learned from these implementations.

In some applications, it is desirable to sacrifice replication transparency and make explicit use of the underlying replication. A scheme based on generators or streams is proposed for this purpose.

The design of a configuration language and manager to handle the programming-in-the-large aspects of constructing programs from troupes is presented.

Chapter 8 summarizes the main points of the dissertation and suggests some areas for future research.

Chapter 2

Background and Related Work

This chapter presents an overview of the areas of reliability, communication, and synchronization, and summarizes the related work in each.

2.1 Reliability

Reliability can mean either *robustness* or *fault tolerance.* Robust systems preserve the consistency of permanent information in the presence of failures, but may become unavailable for some period. Robustness requires a crash recovery algorithm to restore a consistent system state after a crash. Fault-tolerant systems, also called *nonstop* or *highly available* systems, continue to operate correctly in the presence of failures. Fault tolerance requires replication to mask the failures of individual components.

2.1.1 Model of Failures

The hardware components of a distributed computer system consist of processors, storage devices, and networks that connect them. They may fail in various ways. Certain failure modes (classes of misbehavior, perhaps with associated probability distributions) can be identified and incorporated into the specification for the component. Other failures must be relegated to the class of *disasters.*

Processor failures consist of *crashes*, during which a machine simply ceases to function, and *malfunctions*, during which it functions incorrectly. The problem of *byzantine agreement* [78] must be solved in any distributed system whose processors are liable to malfunction.

The byzantine agreement problem is as follows. A sending processor wishes to communicate some value to each of n receiving processors. The sender may malfunction, sending different values to different recipients, or not sending anything to some recipients. Nevertheless, the following two conditions must hold:

1. All correctly functioning recipients agree on the same value.

2. If the sender is functioning correctly, then all correctly functioning recipients agree on the value sent.

This problem was first identified and solved by the researchers who tried to prove the correctness of the SIFT system [78,110,111]. Algorithms for reaching byzantine agreement under various assumptions have since been proposed, and lower bounds on various aspects of the problem have been obtained [51,78,98].

Schneider introduced the notion of a *fail-stop processor*, an idealized machine that may crash but will never malfunction [88,90]. If it is possible to detect malfunctions, then an ordinary processor can be transformed into a fail-stop processor by causing it to halt whenever a malfunction occurs. Schneider has shown how to construct arbitrarily good approximations to fail-stop processors by using byzantine agreement and ordinary processors [90]. For example, an approximation that will tolerate up to n faulty processors can be constructed from $n + 1$ processors and perfect stable storage. This composite processor will always appear to crash instead of malfunction as long as one of its component processors is still operating correctly. If processors can be chosen so that their probabilities of malfunctioning are independent, then by increasing n, the probability that the constructed processor will behave like an ideal fail-stop processor can be made arbitrarily close to certainty.

2.1.2 Crash Recovery

Modern work on crash recovery was first performed in the context of file systems and database systems, where the need to preserve data consistency across crashes is acute. When a processor crashes, all of its *volatile* information is lost; this usually includes the entire contents of main memory. All crash recovery schemes, therefore, rely on some form of non-volatile storage.

Typical examples of non-volatile, random-access storage devices are magnetic or optical disks. A disk consists of a sequence of *pages* that can be read and written. (Pages on optical disks may only be written once.) The information, once written, is non-volatile. For the purposes of crash recovery, however, this is not enough; if the disk or the processor doing the write operation fails while a page is being written, the validity of the page, and hence the consistency of entire data structures, is in doubt.

The operation of writing a disk page must be *atomic*: even if a crash occurs, the page must appear to have been either correctly written or not written at all. Writing a page can be made atomic with high probability by implementing *stable storage* [53,57]. This involves writing each logical page of data onto more than one disk and modifying the read and crash recovery operations to take advantage of the redundancy. By increasing the degree of replication, the probability that the copies of a disk page can become corrupted in such a way that no consistent read operation is possible can be made arbitrarily small. Stable storage, and indeed any form of non-volatile storage, is thus a probabilistic approximation to the unattainable *perfect stable storage*.

Crash recovery mechanisms use stable storage in two ways: for *checkpoints* and *logs*. A checkpoint is a snapshot of a consistent state that can be restored after a crash. A log is a record of the events or operations that affect the state of the system; it is replayed after a crash. Checkpoints provide faster crash recovery, while logs are less expensive during normal operation. If a combination of these two schemes is used, the log need only be replayed from the most recent checkpoint, and the time between checkpoints can be used to balance the cost of the normal and recovery modes of operation.

2.1.3 Fault Tolerance

The idea of achieving fault tolerance by using replication to mask the failures of individual components dates back to von Neumann [106]. The two architectures for fault-tolerant software are *primary/standby* systems and *modular redundancy*. In a primary/standby scheme, only a single component functions normally; the remaining replicas are on standby in case the primary fails. With modular redundancy, each component performs the same function; there is some form of voting on the outputs to mask failures.

A classic primary/standby architecture is the method of *process pairs* in Tandem's Guardian operating system [3,102]. The processes in a process pair execute on different processors. One process is designated as the primary, the other as the standby. Before each request is processed, the primary sends information about its internal state to the standby, in the form of a checkpoint. The checkpoint enables the standby to complete the request if the primary fails.

The Auragen architecture combines a primary/standby scheme with automatic logging of messages [12]. If a primary crashes, the log is used to replay the appropriate messages to a standby.

The Isis project at Cornell uses a primary/standby architecture for replicated objects [5,6,7,8]. In each interaction with a replicated object in Isis, one replica plays the role of *coordinator*, and only it performs the operation. The coordinator then uses a two-phase commit protocol to update the other replicas.

The mechanisms used in primary/standby schemes to allow a standby to take over after the primary crashes are isomorphic to the crash recovery mechanisms described in Section 2.1.2. Under this isomorphism, a standby corresponds to stable storage while the primary continues to function, but assumes the role of the recovering machine when the primary fails.

Another replicated structure for highly available distributed computing is the *worm* [93]. A worm starts by acquiring a set of processors willing to act as *segments* and initializes them with a program to execute. Each segment remains in contact with the others, so that if one becomes unreachable due to a processor or network failure, the worm can regenerate itself by finding a replacement for the unreachable segment.

Each client of a worm-based service must be prepared to direct its requests to a different machine if the segment it is using crashes. A worm is therefore a primary/standby architecture, but with no application-level crash recovery algorithm for state restoration. The service as a whole is highly available, but the probability that an individual request will be completed is determined by the reliability of the particular segment to which it is sent. If this is unacceptable, as is the case in applications where each operation must be highly reliable, then other methods must be used.

Triple-modular and N-modular redundancy have long been familiar to designers of fault-tolerant computer systems [2,66]. Early applications of modular redundancy to software fault tolerance include the SIFT system [110,111] and the PRIME system [26]. Triple modular redundancy with majority voting was initially used in the SIFT system. In this scheme, every computation is carried out by each of three processors. The results are then compared, and if at least two agree, that value is used. The intention was that the voting would detect and correct any single processor malfunction, but the SIFT implementors later realized that byzantine agreement is required in order to guarantee that the replicated components all receive the same information. If malfunctions are assumed not to occur, then voting is unnecessary; only crash detection is required.

Replication is also the basis of methods proposed by Lamport [49] and Schneider [88,90] for constructing distributed systems that meet given reliability requirements.

Gifford's weighted voting scheme uses quorums and version numbers to provide replication transparency for files [30]. Herlihy applied Gifford's quorums to replicated abstract data types [37] by taking advantage of the particular semantics of the data types.

Gunningberg's design of a fault-tolerant message protocol based on triple-modular redundancy [33] is similar to, but less general than, the replicated mechanisms presented in this dissertation.

A methodology known as N-version programming uses multiple implementations of the same module specification to mask software faults [13]. This technique can be used in conjunction with the replicated modules proposed in the present work by using independently implemented modules instead of exact replicas, thereby increasing software as well as hardware fault tolerance. The problems posed by incorrect software are not otherwise addressed in this research.

2.2 Communication

A network enables processors to communicate by sending *packets*: messages with some fixed maximum length. Packets are unreliably delivered; they may be lost, delayed, duplicated, or garbled. Sufficient use of redundancy, in the form of a checksum, can transform garbled packets into lost packets with arbitrarily high probability. It is therefore assumed that packets, when they arrive at all, arrive intact.

The implementations of most current local-area networks permit packets to be addressed to multiple recipients [11,22,67]. This is called *broadcasting* or *multicasting*. It is possible to improve the performance of some algorithms if broadcasting is available, but this is not essential to their correctness. No additional assumptions are made about broadcast packets; in particular, the reliability of delivery may vary from recipient to recipient.

2.2.1 Remote Procedure Call

Remote procedure call (RPC) enables programmers to write distributed programs in the same style as conventional programs for centralized computers [10,73]. To call a remote procedure, the name of the procedure and its parameters are sent in a message to the remote processor, which executes the specified procedure and returns its results in another message. Details of communication are hidden, and the syntax of a remote call is identical to that of a local call. The programmer does not need to know the mapping of modules to machines; indeed, this mapping may be controlled by the system and perhaps changed while the program is running. Therefore, an important requirement for any implementation of remote procedure call is that the semantics of remote calls be as similar as possible to the semantics of the local case. A complete remote procedure call facility must address the following issues.

- Parameter, result, and exception passing semantics.
- External representation of data types.
- Reliable communication of call and return messages of variable length.
- Procedure invocation semantics.
- Binding semantics.

The protocols implemented in the course of this research began as an attempt to transfer the Courier remote procedure call protocol [115] and the Xerox PARC RPC ideas [10,74,73] to an environment based on the UNIX[1] operating system [43] and DARPA Internet protocols [80,81,82].

Sun Microsystems has proposed a remote procedure call protocol that includes a facility for *broadcast RPC* [100], and Cheriton and Zwaenepoel have studied *one-to-many communication* in the context of the V system [14]. These types of communication are equivalent to a special case of replicated procedure calls: the one-to-many calls discussed in Section 4.3.

2.3 Synchronization

A process is a context for a sequential computation; it may be viewed as an abstraction of a processor executing only that computation. Several processes may execute concurrently (on a multiprocessor), or they may be arbitrarily interleaved (to provide the illusion of concurrency on a single processor). In order to avoid chaos when multiple processes access the same data, some form of synchronization is required. The *monitor* is an important synchronization construct that provides mutual exclusion as well as packaging together the shared data and the operations on it in a modular fashion [39,54].

2.3.1 Transactions in Database Systems

The use of *transactions* in database systems is another means of structuring concurrent computations [25,31,104]. Transactions have two essential properties: *atomicity* and *serializability*.

Atomicity guarantees that a transaction is an all-or-nothing operation; no intermediate effects of a transaction are ever visible to other transactions. Until a transaction terminates, its updates are *tentative*. A transaction terminates successfully by *committing*, or unsuccessfully by *aborting*. If a transaction commits, its tentative updates become permanent and visible to other transactions. If a transaction aborts, its tentative updates are undone, leaving no trace of ever having been performed. The fact that tentative updates are not visible to other transactions means that aborts never cascade: when a transaction aborts, no other transaction, either still running or already committed, could have relied on the updates performed by the aborted transaction. Providing atomicity when more than one machine is involved requires some form of *distributed commit protocol* [17], the best known of which is *two-phase commit* [31,53,57].

Serializability means that the concurrent execution of any number of transactions is equivalent to their serial execution in some order. If each transaction transforms a consistent state into another consistent state, then this property ensures that overall consistency is preserved when transactions execute concurrently.

Achieving serializability is the responsibility of a *concurrency control algorithm*. As Bernstein and Goodman [4] demonstrate, most concurrency algorithms use *two-phase locking* [25], *time stamps* [84], or commit-time *validation* [48].

The simplest version of two-phase locking associates a lock with each shared object. One transaction at a time can acquire a lock, hold it for some period, and finally release it. A transaction must hold an object's lock before it can perform any operation on that object. If a transaction attempts to acquire a lock already held by another transaction, the

[1]UNIX is a trademark of Bell Laboratories.

requesting transaction must wait until the lock is released. Serializability is guaranteed by a locking protocol that requires each transaction to hold all locks it has acquired until it either commits or aborts. More sophisticated versions of two-phase locking use different types of locks for the different operations that can be performed on objects. A lock for a particular operation on an object can be acquired as long as no other transaction holds a lock for a conflicting operation on the same object, thus allowing operations that do not conflict with one another to proceed concurrently.

Define the relation T *waits for* T' to be true when transaction T waits for a lock held by transaction T'. A cycle in the *waits for* relation is called a *deadlock*; the transactions involved will wait forever. Several algorithms have been developed to detect deadlock in distributed systems [31,75]. To break a deadlock once it has been detected, any transaction in the cycle may be aborted and restarted.

Time stamps are another method of synchronizing transactions. Each transaction is given a unique time stamp, and each object records the time stamp of the last transaction that operated on it. Suppose a transaction with time stamp τ attempts to perform an operation on an object with time stamp τ'. If $\tau \geq \tau'$, the transaction is allowed to proceed; otherwise, it is aborted. The resulting serialization order is thus identical to the chronological order of the transactions.

Commit-time validation, also called optimistic concurrency control, does not synchronize operations at all, under the optimistic assumption that conflict is unlikely to occur. Instead, the history of operations performed by a transaction is checked at commit time. If serializability would be violated by committing the transaction, it is aborted instead.

Transactions have also been used in file servers and object storage systems [40,99,101], operating systems [96], and programming languages [63,64].

2.3.2 Transactions in Programming Languages

Monitors alone are insufficient as a basis for atomic transactions in programming languages. The mutual exclusion provided by monitors is not enough to ensure serializability, because unless a monitor locks the resources of any monitors it calls, it may be indirectly affected by the actions of other processes that also call those monitors. In order to handle aborts, it must be possible to undo the partial effects of a procedure within a monitor and release its locks. The implementation of monitors in Mesa [54], for example, requires the programmer to do this explicitly by using the exception handling mechanism.

The mutual exclusion provided by monitors also limits concurrency. The work on atomic abstract data types by Allchin [1], Schwarz and Spector [91] and Weihl [107,108,109] takes advantage of the particular semantics of data types, in order to increase concurrency and improve performance.

The use of nested transactions at the programming language level was introduced by Lomet [64] and more fully developed by the Argus project at MIT [61,62,63]. Nested transactions generalize single-level atomic transactions in a way that allows them to mesh properly with the constructs for composition and abstraction supported by programming languages.

A nested transaction consists of a tree of subtransactions, with a single top-level transaction at the root. The tentative updates of a transaction that has not yet committed are visible only to its descendants in the tree. The effects of a committed subtransaction are visible only to ancestors and siblings in the tree. If a transaction aborts, then any uncommitted subtransactions must be aborted, and the effects of any committed subtransactions must be undone.

Reed proposed the first algorithm for nested transactions, using multiple time-stamped versions of objects [84,85]. Moss developed another algorithm based on two-phase locking [70,71].

Chapter 3

A Model of Replicated Distributed Programs

This chapter introduces a program model consisting of modules and threads of control. Modules are used to express the static structure of a program when it is written. Threads of control are manifestations of the dynamic structure of a program when it is executed.

A formal model is used first to treat the pattern of control flow within threads, and then to explore state transitions within modules. The model is then used to unify these dual aspects of program semantics (execution histories and state transitions) in the important case of deterministic programs.

This chapter also shows how to implement distributed but unreplicated versions of modules and threads in terms of primitives provided by current systems.

Troupes are introduced as a means of masking partial failures, and conditions that guarantee replication transparency for programs constructed from troupes are presented.

3.1 Modules

A module is a programming language construct that facilitates the construction of large programs. It packages together the procedures and state information needed to implement a particular abstraction, and separates the *interface* to that abstraction from its *implementation.*

The interface to a module should specify its semantics completely, so that a programmer need only understand the interface in order to make use of an implementation. The interface enforces *information hiding* [77] by encouraging programmers to write code that depends only on the interface, not on details of a particular implementation.

Ideally, an interface should specify the semantics of a module formally enough so that a particular implementation could be checked by machine for conformance to its specification. A complete test is impossible, since conformance to a specification is a generalization of the halting problem and hence undecidable, so a simpler, decidable test must be used. In current programming languages, such as Mesa [68], Ada [105], and Modula-2 [113], an interface consists of declarations of procedures (and related constants, types, and exceptions), and implementations of the interface are checked for type compatibility with these declarations.

The state information required to implement an abstraction is represented by variables that are visible only inside the implementation, not in the interface. This causes no loss of expressive power, since it is a simple matter to define procedures for reading and writing the contents of private variables. It has the desirable consequence that the only flow of information into and out of a module is through a purely procedural interface; changes in the state of a module occur only as side effects of the execution of its procedures.

For simplicity, the state information of a module will be referred to as a single state variable. This results in no loss of generality, since the single state variable may be a structure with multiple components.

A module is said to *export* the procedures defined in its interface. A module that calls procedures in another interface is referred to as a *client* of that interface and is said to *import* that interface.

3.1.1 Comparison of Modules with Abstract Data Types

Modules are similar to abstract data types. Both notions package the concrete representation of an abstraction together with the implementation of the operations defined on it, and present clients with a procedural interface that is separate from the implementation. The difference is that there is only one logical instance of a module (with a single state variable), while there are typically many instances of an abstract data type (each with its own state variable).

This dissertation discusses troupes and replicated procedure call in the context of modules, but these concepts apply equally well to instances of abstract data types.

3.2 Threads

A thread of control is an abstraction intended to capture the notion of an active agent in a computation. A program begins execution as a single thread of control; additional threads may be created and destroyed either explicitly by means of **fork**, **join**, and **halt** primitives [16,54], or implicitly during the execution of a **cobegin** ... **coend** statement [23].

Each thread is associated with a unique identifier, called a *thread ID*, that distinguishes it from all other threads.

A particular thread runs in exactly one module at a given time, but any number of threads may be running in the same module concurrently. Threads move among modules by making calls to, and returning from, procedures in different modules. The control flow of a thread obeys a last-in first-out (or stack) discipline.

3.2.1 Comparison with Earlier Work

The **fork**, **join**, and **halt** primitives for lightweight processes (single-machine threads of control) were first described by Conway [16].

The ability for a process to cross address spaces when calling a procedure was supported in hardware by the Burroughs B6500 stack architecture [15].

The module and thread abstractions presented here are most similar to the domains and threads of the Trix kernel [94]. In Trix, threads move among domains by means of "remote" procedure calls, but the domains must be address spaces on a single machine.

3.3 Semantics of Modules and Threads

The behavior of a thread is expressed in terms of its history of procedure calls and returns. The model introduced in the following definitions captures the last-in first-out pattern of control flow in threads.

3.3.1 Flow of Control in Threads

The set of procedures exported by a module M is denoted by $\textbf{Procs}(M)$. If P is a procedure, $\textbf{module}(P)$ denotes the unique module M such that $P \in \textbf{Procs}(M)$.

An *event* is a call to or a return from a procedure, and is represented as a quadruple of the form (op, proc, val, id), where op is the operation (either call or return), proc is the procedure, val is the list of values being passed to or returned from the procedure, and id uniquely identifies this event. The expressions op(e), proc(e), val(e), and id(e) will be used to denote the components of an event e. The expression module(e) will be used as an abbreviation for module(proc(e)).

An *event sequence* E is an ordered set of distinct events $\langle e_0, e_1, \ldots, e_i, \ldots \rangle$. Event sequences will be used to represent computations; since some computations never terminate, event sequences may be infinite. The ordering on E is denoted by $e_0 < e_1 < \cdots < e_i < \cdots$. The *successor function* succ(e) is defined on all events $e \in E$ (except the final event in the case that E is finite). The *predecessor function* pred(e) is defined on all events $e \in E$ except the initial event.

A *subsequence of* E is any subset of E together with the inherited ordering. If $\varphi(e)$ is a predicate on events, the expression $\langle e \in E \mid \varphi(e) \rangle$ denotes the subsequence of E consisting of those events e for which $\varphi(e)$ is true. Note that a subsequence need not be a contiguous portion of E.

Given two events e_1 and e_2 in E, the *event interval* $\langle e_1, \ldots, e_2 \rangle$ is the subsequence

$$\langle e \in E \mid e_1 \le e \le e_2 \rangle$$

The events e_1 and e_2 are called the left and right endpoints of the interval.

Let e be an event in E. The *portion of E up to e*, denoted $E_{\le e}$, is the subsequence defined by

$$E_{\le e} = \langle e' \in E \mid e' \le e \rangle$$

An event e with module(e) $= M$ is called an *M-event*. The *restriction of an event sequence* E to a module M, denoted E^M, is the subsequence of M-events in E:

$$E^M = \langle e \in E \mid \text{module}(e) = M \rangle$$

Since $(E_{\le e})^M = (E^M)_{\le e}$, the notation $E^M_{\le e}$ is unambiguous.

The *concatenation* of two disjoint sequences E_1 and E_2 is the sequence (denoted $E_1 E_2$) defined by the union of E_1 and E_2 together with the ordering in which E_1 "comes before" E_2. Concatenation is an associative operation, so the expression $E_1 \cdots E_n$ is unambiguous.

Definition 3.1. An interval $B = \langle c, \ldots, r \rangle$ of length ≥ 2 is *balanced* if c is a call, r is a return, proc(c) = proc(r), and for some $n \ge 0$,

$$B = \langle c \rangle B_1 \cdots B_n \langle r \rangle$$

where each B_i is balanced. (It follows that $B_1, \ldots, B_n$ are uniquely determined.)

A call $c \in E$ is said to *return at* r if the interval $\langle c, \ldots, r \rangle \subseteq E$ is balanced. The *execution of a call* c, denoted Exec(c), is the event sequence defined by

$$\text{Exec}(c) = \begin{cases} \langle c, \ldots, r \rangle & \text{if } c \text{ returns at } r \\ \langle e \in E \mid e \ge c \rangle & \text{if } c \text{ never returns} \end{cases}$$

The following are equivalent:

1. $\langle c, \ldots, r\rangle$ is a balanced interval
2. c returns at r
3. $\mathbf{Exec}(c) = \langle c, \ldots, r\rangle$

Definition 3.2. A *thread execution history* H is an event sequence that satisfies the following conditions:

1. Every return $r \in H$ determines a unique call $c \in H$ that returns at r.
2. If H is finite, then H is balanced.

It follows from this definition that for any thread execution history H with initial event e_0,

1. e_0 is a call,
2. $H = \mathbf{Exec}(e_0)$, and
3. H is finite if and only if every call returns.

Definition 3.3. The *call stack* after a call c, denoted $\mathbf{Callstack}(c)$, is the event sequence consisting of all calls $c' \leq c$ that do not return before c. Since a call $c' \leq c$ does not return before c if and only if the execution of c' contains c,

$$\mathbf{Callstack}(c) = \langle c' \mid c \in \mathbf{Exec}(c')\rangle$$

Alternatively, $\mathbf{Callstack}(c)$ can be obtained from $H_{\leq c}$ by removing all balanced intervals. Clearly, the events that remain are precisely the calls in $H_{\leq c}$ that do not return before c. The *depth* of a call $c \in H$, denoted $\mathbf{depth}(c)$, is defined to be the length of $\mathbf{Callstack}(c)$.

The following theorem is a useful characterization of the structure of thread execution histories.

Theorem 3.4. $H_{\leq e}$ can be written uniquely in the form

$$\langle c_0, \ldots, c\rangle B_1 \cdots B_n \langle e\rangle$$

where c_0 is the initial call in H, c is a call such that $c_0 \leq c < e$, and $B_1, \ldots, B_n$ are balanced intervals for some $n \geq 0$.

Proof: If e is a return, then by the first part of Definition 3.2, there is a unique call $c \in H$ that returns at e. Since $\langle c, \ldots, e\rangle$ is balanced, the existence and uniqueness of $B_1, \ldots, B_n$ follow from Definition 3.1. If e is a call, define c by

$$c = \begin{cases} c_0 & \text{if } e = c_0 \\ \text{the predecessor of } e \text{ in } \mathbf{Callstack}(e) & \text{if } e \neq c_0 \end{cases}$$

By Definition 3.3, the events between c and e in $H_{\leq e}$ that are elided in $\mathbf{Callstack}(e)$ form the balanced intervals $B_1, \ldots, B_n$. □

Both calls and returns appear explicitly in event sequences so that the structure of balanced intervals is uniquely determined. An equivalent approach is to represent a thread execution history as a *procedure invocation tree*, in which nodes represent procedure executions and edges (appropriately ordered) represent calls. In this model, the root of the tree is the initial call, and each path from the root to an interior node is the call stack for that node.

The tree model, while intuitively appealing, is difficult to manipulate formally and a nightmare to typeset. In fact, the author's attempts at a sufficiently formal linear notation for trees ended up isomorphic to the event sequence model presented here.

3.3.2 State Transitions and Deterministic Modules

A *program state* σ is a mapping that assigns a value σ^M to the state variable of each module M. The set of all possible program states is denoted by Σ.

Definition 3.5. A *state sequence* for a thread execution history H is a function

$$\mathbf{state}\colon H \longrightarrow \Sigma$$

with the property that for each module M, only M-events affect the state of M. This means that if $e' = \mathrm{succ}(e)$ and either

1. e is not an M-event and e' is a call to a procedure in M, or
2. e is a return from a procedure in M and e' is not an M-event, or
3. neither e nor e' is an M-event,

then $\mathbf{state}^M(e) = \mathbf{state}^M(e')$. The expression $\mathbf{state}(e)$ represents the program state at the time of e, after the events in $H_{\leq e}$ have occurred. The initial state is denoted by $\mathbf{state}(\emptyset)$. The notation $\mathbf{state}_H(e)$ will be used when necessary to avoid confusion.

Definition 3.6. A module M is *deterministic* if for any call c to a procedure in M and any interval I of the form

$$I = \langle c \rangle B_1 \cdots B_n$$

where $n \geq 0$ and each B_i is balanced, I and $\mathbf{state}^M(c)$ uniquely determine the event that follows I and the state of M when that event occurs. Note that if $\mathrm{proc}(c) = P$, then I is any prefix of $\mathrm{Exec}(c)$ that stops just before a call made by P or the return from P corresponding to c.

Determinism is a property local to a single module: it constrains only the events under the control of the procedures in that module. It is tempting to demand that the entire execution of a call in a deterministic module be uniquely determined, rather than just the events one level deeper than the call, but this would require that all other modules called during the execution also be deterministic. Definition 3.6 allows a deterministic module to call nondeterministic modules, so that a program can be composed of both kinds.

In a sense, a nondeterministic module "contaminates" any module that imports it, and makes the program as a whole nondeterministic. The notion of *global determinism* is therefore useful. A program is globally deterministic if it consists entirely of deterministic modules. Global determinism is an extremely strong property, as is shown in the next theorem.

Theorem 3.7. *Let H be an execution history for a thread in a globally deterministic program and let $\mathbf{state}_H$ be a state sequence for H. Then the initial call c_0 in H and the initial program state $\mathbf{state}_H(\emptyset)$ suffice to determine H and $\mathbf{state}_H$ uniquely.*

Proof: Let H and K be two thread execution histories

$$H = \langle e_0, e_1, \ldots \rangle \quad \text{and} \quad K = \langle e'_0, e'_1, \ldots \rangle$$

where $e_0 = e'_0 = c_0$, and let $\mathbf{state}_H$ and $\mathbf{state}_K$ be state sequences for H and K with the same initial state. It will be shown by induction that $e_n = e'_n$ and $\mathbf{state}_H(e_n) = \mathbf{state}_K(e_n)$ for all $n \geq 0$.

The case $n = 0$ is obvious from the hypotheses. So suppose the theorem is true for n. Then

$$H = \langle e_0, e_1, \ldots, e_n, e_{n+1}, \ldots \rangle \quad \text{and} \quad K = \langle e_0, e_1, \ldots, e_n, e'_{n+1}, \ldots \rangle$$

By Theorem 3.4 with $e_{n+1} \in H$ and $e'_{n+1} \in K$, there is a call c with $c_0 \leq c \leq e_n$ such that

$$H = \langle c_0, \ldots, c \rangle B_1 \cdots B_k \langle e_{n+1} \rangle \quad \text{and} \quad K = \langle c_0, \ldots, c \rangle B_1 \cdots B_k \langle e'_{n+1} \rangle$$

Write B_k as $\langle c_k, \ldots, r_k \rangle$. Then since $\mathbf{module}(c)$ is deterministic, Definition 3.6 implies that $\mathbf{succ}_H(r_k) = \mathbf{succ}_K(r_k)$, so $e_{n+1} = e'_{n+1}$.

Finally, it follows from the induction hypothesis and Definitions 3.5 and 3.6 that $\mathbf{state}_H(e_{n+1}) = \mathbf{state}_K(e_{n+1})$. $\square$

Theorem 3.7 can be viewed as a formal statement and proof of the equivalence of the two crash recovery mechanisms described in Section 2.1.2: restoring a consistent state from a checkpoint, or replaying events from a log.

3.4 Implementing Distributed Modules and Threads

The model presented so far in this chapter defines two abstractions: modules and threads; no mention is made of machine boundaries as part of their semantics. A distributed implementation of these abstractions must therefore provide *location transparency*: the execution history of a distributed program must be indistinguishable from that of a single-machine program. A programmer need not know the eventual configuration of a program when it is being written; the fact that a program is distributed is invisible at the programming-in-the-small level.

A module in a distributed program can be implemented by a *server* whose address space contains the module's procedures and data. A distributed thread can be implemented by using remote procedure calls to transfer control from server to server, and viewing such a sequence of remote procedure calls as a single thread of control. Recall that the thread abstraction includes a unique ID for each thread. The thread ID propagation algorithm that follows shows how this aspect of distributed threads can be implemented.

3.4.1 The Thread ID Propagation Algorithm

The call stack of a distributed thread at a given instant can be divided into *segments* at every inter-module call. Each segment consists of a series of calls within the same module.

Call this module the *locus* of the segment. Since a module is never split across machine boundaries, a segment can be represented by the stack of a conventional process running on the same machine as the locus of the segment.

A single conventional process, called a *base process*, is required to represent the segment containing the base of the call stack. The lifetime of the base process is identical to that of the entire distributed thread, so its local process ID together with a machine ID can be used as a unique thread ID.

The remote procedure call mechanism is responsible for creating a server process for each incoming call to a server, and destroying that process once execution of the procedure is complete. (The expense of process creation may be reduced by maintaining a pool of idle server processes, but the effect in either case is to create and destroy the segment of the call stack required for the execution of the procedure.) These server processes can therefore be used to represent the remaining segments of the call stack of the distributed thread.

Finally, the remote procedure call mechanism must be extended to propagate thread IDs from client processes to server processes. Each call message bears the thread ID of the caller, and each server process assumes this thread ID while it is performing the procedure requested by the call message. This scheme effectively makes the current thread ID an extra parameter of every remote procedure.

It follows that at any instant,

1. all the processes representing segments of the call stack of a distributed thread bear the same thread ID, and

2. the thread ID uniquely identifies the distributed thread.

This algorithm therefore correctly associates a unique ID with each thread as it moves through the distributed system by means of remote procedure calls.

3.5 Adding Replication

The distributed modules and threads of Section 3.4 provide location transparency in the absence of failures. As long as the underlying hardware works correctly, the programmer need not be aware of machine boundaries.

Processor and network failures, however, give rise to new classes of *partial failures* of the distributed program as a whole. Partial failures violate transparency, since they can never occur in a single-machine program. These failures must therefore be masked if transparency is to be preserved.

The key to masking failures is replication, but it introduces another transparency requirement: *replication transparency*.

3.5.1 Troupes

The approach taken in this research is to introduce replication into distributed programs at the module level. A replicated module is called a *troupe*, and the replicas are called *troupe members*.

Troupe members are assumed to execute on fail-stop processors [88,90]. If the processors were not fail-stop, troupe members would have to reach byzantine agreement about the contents of incoming messages, because a malfunctioning processor might send different messages to different troupe members. Byzantine agreement, described in Section 2.1.1, could be added to the algorithms presented in this dissertation, but would result in a significant loss of performance. There is no evidence that failures other than crashes occur often enough to warrant this increased expense.

A *deterministic troupe* is a set of replicas of a deterministic module (as defined in Section 3.3.2). Section 3.5.2 shows that the assumption that all troupes are deterministic is sufficient to guarantee replication transparency.

In contrast to the work on replicated abstract data types by Herlihy [37] and on atomic abstract data types by Allchin [1], Schwarz and Spector [91], and Weihl [107,108,109], troupes are a simple approach to achieving high availability: no knowledge of the semantics of a module is required, other than the fact that it is deterministic.

Interactions between troupes occur by means of replicated procedure calls (discussed in Chapter 4) in which all troupe members play identical roles. Furthermore, troupe members do not know of one another's existence; there is no communication among the members of a troupe. It follows that each troupe member behaves exactly as if it had no replicas. In this sense, troupes contrast sharply with the replicated objects in Isis [5,6,7,8], although the goal of high availability is the same.

In replicated distributed programs, the crash recovery mechanisms of Section 2.1.2 are required only for total failures, in which every troupe member crashes. The analysis in Section 6.4.2 shows that the probability of total failures can be made arbitrarily small by choosing an appropriate degree of replication. Replication can therefore be used as an alternative to crash recovery mechanisms such as stable storage.

3.5.2 Replication Transparency and Troupe Consistency

A troupe is *consistent* if all its members are in the same state. If a troupe is consistent, then its clients need not know that it is replicated. Troupe consistency is therefore a sufficient condition for replication transparency.

Troupe consistency is a strong requirement, but it cannot be weakened without knowledge of the semantics of the objects being replicated. *In the absence of application-specific knowledge, troupe consistency is both necessary and sufficient for replication transparency.* This is one area in which troupes differ from other replication schemes. Gifford's weighted voting for replicated files, for example, uses quorums and version numbers to mask the fact that not all replicas are up to date [30], and Herlihy has extended Gifford's approach to abstract data types [37]. Troupe consistency is not necessary in these schemes, because they take advantage of the semantics of the objects being replicated.

In a program constructed from troupes, an inter-module procedure call results in a replicated procedure call from a client troupe to a server troupe, as described in Chapter 4. One of the distinguishing characteristics of troupes is that their members do not communicate among themselves, and do not even know of one another's existence. Consequently, when a client troupe makes a replicated call to a server troupe, each server troupe member must perform the procedure, just as if the server had no replicas.

As mentioned in Section 3.3.1, the execution of a procedure can be viewed as a tree of procedure invocations. The global determinism property of Section 3.3.2 implies that when a server troupe is called upon to execute a procedure, the invocation trees rooted at each troupe member are identical: the members of the server troupe make the same procedure calls and returns, with the same arguments and results, in the same order. It follows from Theorem 3.7 that if there is only a single thread of control in a globally deterministic replicated distributed program, and if all troupes are initially consistent, then all troupes remain consistent.

Additional mechanisms are required if there is more than one thread of control, because concurrent calls to the same server troupe may leave the members of the server troupe in inconsistent states. The problem of maintaining troupe consistency in the presence of concurrently executing threads is addressed in Chapter 5.

Chapter 4

Replicated Procedure Calls

A thread in a replicated distributed program transfers control from one troupe to another by means of replicated procedure calls. This chapter defines the semantics of replicated procedure calls and describes Circus, a replicated procedure call system that has been implemented for Berkeley 4.2BSD.

Replicated procedure calls are implemented on top of a paired message layer that exchanges variable-length messages reliably. The paired message protocol used in Circus is described and compared to a similar protocol developed at Xerox PARC.

Replication transparency requires exactly-once execution of each procedure at all replicas. The general replicated procedure call algorithm factors naturally into two subalgorithms, for one-to-many and many-to-one calls.

The performance of the Circus implementation is measured and analyzed, and a theoretical analysis is used to predict the performance of a more efficient implementation.

4.1 Semantics of Replicated Procedure Calls

The goal of remote procedure call [73] is to allow distributed programs to be written in the same style as conventional programs for centralized computers. Details of communication are hidden, and the syntax of a remote call is similar or identical to a normal local call.

When modules are replaced by troupes, the natural generalization of remote procedure call is *replicated procedure call.* The troupe consistency requirement identified in Section 3.5.2 determines the semantics of replicated procedure call: when a client troupe makes a replicated procedure call to a server troupe, each member of the server troupe performs the requested procedure exactly once, and each member of the client troupe receives all the results. These semantics can be summarized as *exactly-once execution at all troupe members.* Figure 4.1 shows a replicated procedure call from a client troupe to a server troupe. A replicated distributed program constructed in this way will continue to function as long as at least one member of each troupe survives.

To guarantee replication transparency, troupe members are required to behave deterministically: two replicas in the same state must execute the same procedure in the same way. In particular, they must call the same remote procedures in the same order, produce the same side effects, and return the same results. Section 7.4 shows how programmers can replace "same" by an application-specific equivalence relation. Note that determinism is also required in the crash recovery schemes described in Section 2.1.2.

Just as replicated procedure call is a conceptual extension of remote procedure call, the Circus implementation described here evolved as an extension of a remote procedure call implementation for Berkeley 4.2BSD. When the degree of module replication is one, Circus functions as a conventional remote procedure call system.

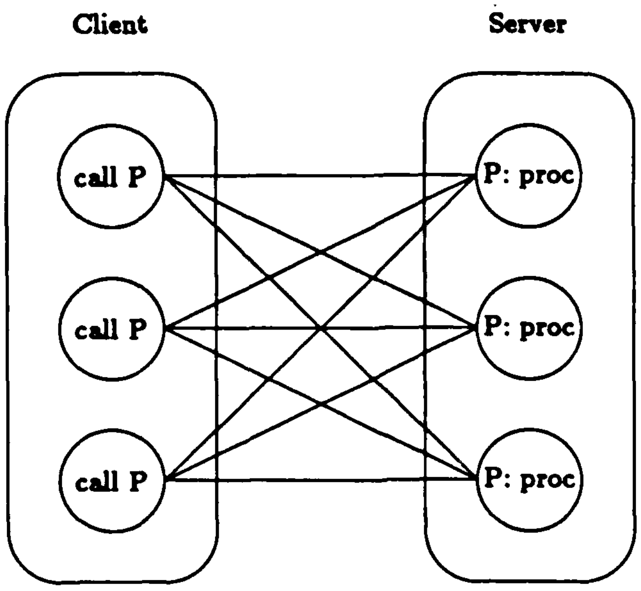

Figure 4.1: Replicated procedure call

4.2 Paired Message Protocols

A *paired message protocol* is a distillation of the communication requirements of conventional remote procedure call protocols [10,73,115]. It provides

- reliably delivered, variable-length, paired messages (e.g. call and return), and
- *call sequence numbers* that uniquely identify each pair of messages among all those exchanged by a given pair of processes.

The paired message protocol is responsible for segmenting messages that are larger than a single datagram (in order to permit variable-length messages), and for retransmission and acknowledgment of message segments to ensure reliable delivery. The Circus protocol presented here is based on the RPC protocol of Birrell and Nelson [10]. Circus uses UDP, the DARPA User Datagram Protocol [80]. The Circus protocol is connectionless and geared towards the fast exchange of short messages. The main difference between it and the PARC RPC protocol lies in the treatment of messages requiring multiple datagrams; the Circus protocol provides better recovery from lost datagrams in this case, at the cost of increased buffering. The two protocols are compared in more detail in Section 4.2.5.

The paired message abstraction can be provided on top of reliable stream protocols like TCP [82] and SPP [114], but implementations of these protocols are typically tuned for bulk data transfers. The Berkeley 4.2BSD implementation of TCP, for example, does not even begin to transfer data until the connection has been established by a three-way handshake, although this restriction is not inherent in the protocol specification. Since call and return messages are usually short, a specially designed, datagram-based paired message protocol like Circus can complete a message exchange using the same number of packets that a stream protocol requires merely to establish a connection. Nelson makes this same point, with performance measurements to support his claim, in his dissertation [73].

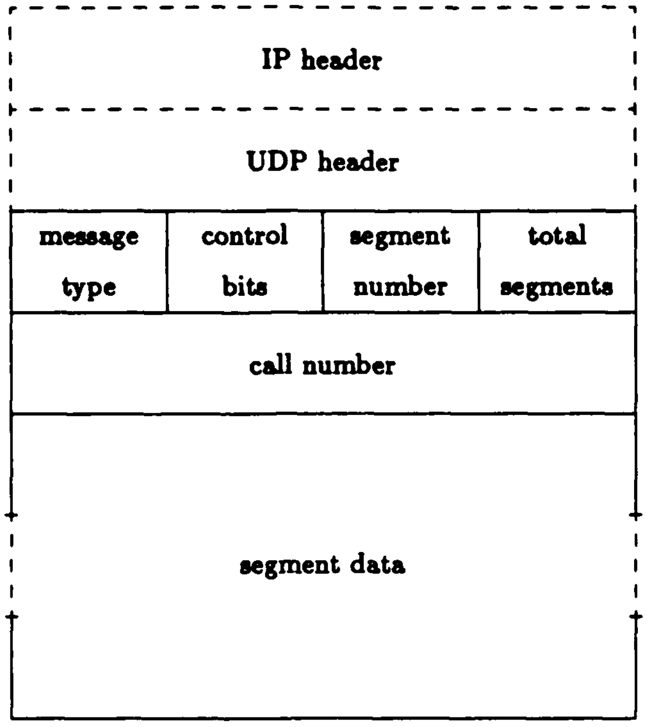

Figure 4.2: Segment format

The paired message protocol does not specify how remote modules or procedures are identified, how clients and servers are bound together, how parameters and results are represented, or how exceptional conditions are handled. The contents of the messages are uninterpreted. It is therefore possible for several remote (or replicated) procedure call systems with different type representation and module binding requirements to use the same paired message protocol as a basis for communication. For example, the Lisp remote procedure call facility described in Section 7.1.3 uses the same paired message protocol as the Circus system, but represents procedures and values symbolically in messages.

4.2.1 Messages

Messages are exchanged between conventional, single-machine processes. A *process address* consists of a 32-bit host address together with a 16-bit port number. The host address identifies the machine within the DARPA internet, and the port number identifies the process within the machine. This is the same address format used by the underlying UDP layer; the assignment of port numbers to processes is left to the UDP implementation.

A message is a sequence of bytes, together with a type (call or return). Messages are transmitted as one or more *segments* of fixed maximum length. A segment is a UDP datagram of the form shown in Figure 4.2.

A *data segment* consists of a segment header together with some portion of the message data. A *control segment* contains only a segment header; it is used to send or request acknowledgment information. The *message type* is a byte containing either 0 for a call message or 1 for a return message. The *control bits* field is a byte containing two Boolean values used to control the acknowledgment and retransmission procedures. The least significant bit is the please ack flag, and the next least significant bit is the ack flag. The six most significant bits are unused.

The next two bytes are used to specify the logical position of the segment within the whole message. The *total segments* field is a byte containing the total number of segments in the message, which must be in the range from 1 to 255, inclusive. The *segment number* is a byte containing a number between 0 and the total number of segments, inclusive. The meaning of this field depends on whether the segment contains data or acknowledgment information.

The *call number* is a 32-bit unsigned integer, represented most significant byte first. The call number is used to pair call messages with the corresponding return messages.

4.2.2 Sending and Receiving Messages

The protocol for sending a message is the same for both client and server; the only difference is whether the message type is call or return. A sequence of bytes to be sent as a message is first divided into segments. Each segment is assigned a number, starting at 1, which is placed in the segment number field of its header. The message type, total number of segments, and call number fields of the header are the same for each segment of the message. The sender maintains a queue of the unacknowledged segments of the message, initially containing all the segments.

The sender initially transmits all the segments to the receiver with no control bits set. It then periodically retransmits the first unacknowledged segment on its queue, with the please ack bit set. Simultaneously, the sender listens for acknowledgments and removes acknowledged segments from its queue. When the queue is empty, all segments have been acknowledged and transmission of the message is complete.

An acknowledgment is either explicit or implicit. An explicit acknowledgment is a segment with the ack bit set and the same message type, call number, and total number of segments as the current message. Acknowledgment segments contain no data; the segment number field is used as an acknowledgment number, indicating that all segments with numbers less than or equal to the acknowledgment number have been received.

An implicit acknowledgment is a data segment sent by the receiver back to the sender. A segment from a return message implicitly acknowledges all the segments of the previous call message if it carries the same call number, and a segment from a call message implicitly acknowledges all the segments of the previous return message if it carries a later call number. Implicit acknowledgments are possible because processes alternate between sending and receiving.

The protocol for receiving a message is also the same for both client and server. The receiver maintains a queue of incoming segments for the current message, and an acknowledgment number, initially zero. The acknowledgment number is the highest consecutive segment number received. When a segment arrives, it is placed in its proper position in the queue. The segment may have filled a gap in the queue, enabling the acknowledgment number to be advanced. If the please ack bit is set in the incoming segment, an explicit acknowledgment segment is sent with the current value of the acknowledgment number in the segment number field. Reception of the message is complete as soon as all the segments have been received.

4.2.3 Crash Detection

Once a client has sent an entire call message and its receipt has been acknowledged, the client may wait arbitrarily long before the remote procedure finishes and sends back the return message. In order to detect crashes during this interval, the client periodically *probes* the server with a special control segment.

The send and receive protocols guarantee that messages will be communicated correctly in the presence of lost or duplicated datagrams (assuming that any segment retransmitted repeatedly will eventually be received). This assumption does not hold in the event of a crash. In order to detect crashes, an upper bound (or timeout) must be placed on the number of retransmissions with no response before it is assumed that the receiver has crashed. A bound that is too low increases the chance of incorrectly deciding that a receiver has crashed. A bound that is too high introduces a long delay in the detection of true crashes.

4.2.4 Implementation Details

Several optimizations are possible to reduce the number of acknowledgments and retransmissions. For instance, when an out-of-order segment arrives during receipt of a multiple-segment message, the receiver knows that one or more segments have been lost. It should therefore immediately send an explicit acknowledgment for the last consecutively received segment, so that the sender will retransmit the first lost segment, rather than an earlier segment.

When a segment that completes a call message arrives at a server, acknowledgment of the message can be postponed, even if the please ack bit was set, in the hope that the return message will be forthcoming soon enough to serve as an implicit acknowledgment. Subsequent please ack segments should be acknowledged promptly.

The retransmission strategy can be changed to retransmit all the remaining unacknowledged segments rather than just the first, depending on the reliability characteristics of the network.

The protocol is connectionless in the sense that no initial handshake is needed to establish communication; a client merely sends a call message to a server. Clients and servers must maintain state information about active message exchanges (segment queues and acknowledgment numbers). After an exchange has completed, only its call number must be kept, and this may be discarded once sufficient time has passed to guarantee that no delayed segments from the exchange can arrive. This prevents the "replay" of delayed call messages.

The maximum length of a segment can be no larger than the maximum UDP datagram size minus the 8 bytes of segment header. It may be desirable to use a smaller limit in order to prevent fragmentation at the IP level [81]. This requires knowledge of the maximum transmission unit (MTU) for the physical networks of interest (presumably the local area networks expected to be used most often).

The Circus protocol is currently implemented entirely in user code under Berkeley 4.2BSD [43].

Asynchronous events, specifically the arrival of datagrams and the expiration of timers, must be handled in parallel with the activity of the client or server. For instance, a probe may arrive while a server is performing a procedure. If multiple processes sharing the same address space were available under Berkeley 4.2BSD, a separate process could be devoted to listening for incoming segments and handling timers. Since this is not possible, these events are modeled as software interrupts using the signal mechanism, the interrupt-driven I/O facility, and the interval timer [43]. Protection of critical regions is achieved by using system calls that mask and enable interrupts.

The protocol package uses timers to handle retransmission, probing, no-response timeouts, and no-activity timeouts. A general timer package was built on top of the single interval timer for this purpose. It allows a timer to be defined by a timeout interval and a procedure to be invoked upon expiration; any number of timers may be active at the same time.

A project is under way at Berkeley to produce a kernel implementation of a remote procedure call protocol [112]. The initial specification was an unreplicated version of the Circus protocol, but the desire to limit the required amount of kernel buffer space led to a protocol similar to Birrell and Nelson's.

The unifying communication abstraction provided by the Berkeley 4.2BSD kernel is the socket [43]. A socket is an endpoint for process-to-process communication. Each socket has a protocol type that is used to dispatch generic operations like `read` and `write` to the appropriate protocol implementation. The interface to the kernel RPC protocol is by means of a new protocol type (RPC) with two subtypes: client and server. The implementation enforces write-read alternation for client sockets and read-write alternation for server sockets.

4.2.5 Comparison with Related Work

The Circus implementation was strongly influenced by the work on remote procedure call done by Birrell and Nelson at Xerox PARC [10,73]. The difference between their RPC protocol and the Circus protocol lies in the treatment of multiple-segment call and return messages.

The Xerox PARC protocol requires an explicit acknowledgment of every segment but the last. This doubles the number of segments sent, but since there is never more than one unacknowledged segment in transit, only one segment's worth of buffer space is required per connection.

The Circus protocol allows multiple segments to be sent before one is acknowledged, which reduces the number of segments sent to the minimum, but requires an unbounded amount of buffering. An alternate implementation of the Circus protocol could easily bound the amount of buffer space required for a connection by dropping all segments outside a fixed allocation window, and simply requiring the sender to retransmit them. These retransmissions could be reduced by informing the sender of the size of the allocation window; this is precisely what is done in the flow-control mechanisms of reliable stream protocols such as TCP [82] and SPP [114], but since single-segment messages are expected to occur most often in remote procedure calls, these optimizations are probably not worthwhile.

4.3 Implementing Replicated Procedure Calls

Replicated procedure calls are made between troupes, which are sets of modules. Since one server may export several modules, a *module address* is a refinement of the internet process address defined in Section 4.2. A module address consists of a process address together with a 16-bit module number that identifies the module among those exported by that process. In the Circus implementation, the module number is an index into a table of exported interfaces managed by the export procedure.

A troupe is represented at this level as a sequence of module addresses. This representation is returned by the binding agent when a client imports a server troupe.

The contents of a call message can now be described. Remember that this data is uninterpreted by the paired message layer. A call message consists of a header containing the thread ID of the caller, the module number and procedure number of the procedure to be called, and the parameters to be passed the procedure. The thread ID is used to implement the thread ID propagation algorithm of Section 3.4.1. The procedure number is assigned by the stub compiler and is the index of the procedure within the module interface. The header also contains other information described below. The parameters are represented in a standard external form by the routines produced by the stub compiler.

A **return** message consists of a 16-bit header (used to distinguish between normal and error results) and the results of the procedure in the standard external representation.

Replicated procedure calls are implemented on top of the paired message layer. There are two subalgorithms involved in a *many-to-many* call from a client troupe to a server troupe: each client troupe member performs a *one-to-many* call to the entire server troupe, and each server troupe member handles a *many-to-one* call from the entire client troupe.

The algorithms for these two cases are described in the following sections. In Circus, these algorithms are implemented as part of the run-time system that is linked with each user's programs. The run-time system is called by stub procedures that are produced automatically from a module interface; the replicated procedure call algorithms themselves are thus hidden from the programmer. When the algorithms below refer to various client and server actions, the reader should bear in mind that those actions are performed by the protocol routines in the corresponding run-time systems, rather than by the portions of the program written by the user.

4.3.1 One-To-Many Calls

The client half of the replicated procedure call algorithm performs a *one-to-many call* as shown in Figure 4.3. The purpose of the one-to-many call algorithm is to guarantee that the procedure is executed at each server troupe member.

The same call message is sent to each server troupe member, with the same call number at the paired message level. The client then awaits the arrival of the **return** messages from the members of the server troupe. Since the **return** messages bear the same call number, it is a simple matter for the client to collect the proper set of responses.

In the Circus replicated procedure call implementation, the client will normally wait for all the **return** messages from the server troupe before proceeding. The client receives notification if any server troupe member crashes, so it can proceed with the **return** messages from those that are still available. The return from a replicated procedure call is thus a *synchronization point*, after which each client troupe member knows that all server troupe members have performed the procedure, and each server troupe member knows that all client troupe members have received the result. Alternatives to this obviously correct but potentially slow strategy are discussed in Section 4.3.4 below.

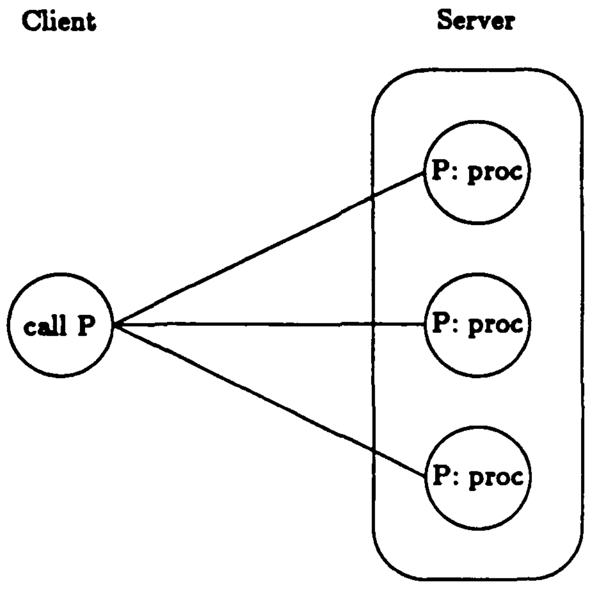

Figure 4.3: A one-to-many call

A one-to-many call could be expressed as several concurrent processes, each performing a conventional remote procedure call, but the present formulation has the advantage of showing how to implement the algorithm using multicast operations [11,22].

4.3.2 Many-To-One Calls

Now consider what occurs at a single server when a client troupe makes a replicated call to it. The server will receive call messages from each client troupe member, as shown in Figure 4.4; this is called a *many-to-one* call. The semantics of replicated procedure call require the server to execute the procedure only once and return the results to all the client troupe members. The many-to-one call algorithm must therefore solve the following two problems:

1. The server must be able to distinguish unrelated call messages from ones that are part of the same replicated call.

2. When one call message of a replicated call arrives, the server must be able to determine how many other call messages to expect as part of the same replicated call.

The solution to the first problem follows from the assumption that troupes are deterministic: two or more call messages arriving at a server bear the same thread ID and call sequence number if and only if they are part of the same replicated call.

To see why, observe that if two call messages are part of the same replicated call, then the client troupe members that sent them are acting on behalf of the same logical thread of control, so the thread ID propagation algorithm (described in Section 3.4.1) attaches the same thread ID to each call message. Since the troupe is deterministic, the

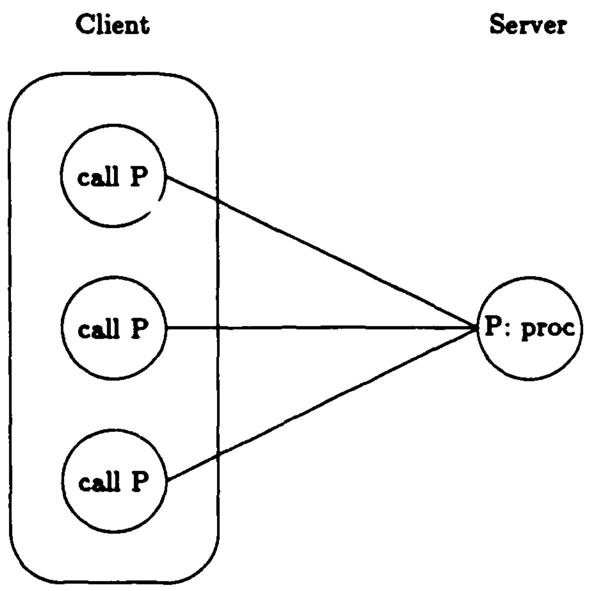

Figure 4.4: A many-to-one call

same call sequence number is used by each client troupe member. Conversely, if two call messages represent distinct procedure calls, then either different logical threads of control are making the calls, in which case the call messages will bear different thread IDs, or they are different calls from the same logical thread of control, in which case the call messages will bear different call sequence numbers.

The solution to the second problem requires a unique ID for each troupe (assigned by the binding agent described in Chapter 6) and an additional field in the call message header containing the client troupe ID. When a server receives a call message, it maps the client troupe ID into the set of module addresses of the members of the client troupe. This is done by consulting a local cache or by contacting the binding agent. In this way, the server learns how many call messages to expect as part of the many-to-one call.

The client troupe ID and thread ID allow the server to collect the entire set of call messages that form a many-to-one call. The procedure is executed once, and a return message containing the results is sent to each member of the client troupe. The server adopts the thread ID in the call header as its own for the duration of the procedure execution (as described in Section 3.4.1) so that the thread ID will be correctly propagated if any further remote calls are made during the procedure.

In Circus, the server waits for call messages from all available client troupe members before proceeding. Alternatives to this strategy are discussed in Section 4.3.4 below.

4.3.3 Many-To-Many Calls

In general, a replicated procedure call is a *many-to-many* call from a client troupe to a server troupe, as shown in Figure 4.1. A many-to-many call involves the following steps:

1. Each client troupe member sends a call message to each server troupe member.

2. Each server troupe member receives a call message from each client troupe member.

3. Each server troupe member performs the requested procedure.

4. Each server troupe member sends a **return** message to each client troupe member.

5. Each client troupe member receives a **return** message from each server troupe member.

The key to the many-to-many case is the observation that steps 1 and 5 are the same steps that an unreplicated client performs when making a one-to-many call to a server troupe, and steps 2, 3, and 4 are the same steps that an unreplicated server performs when handling a many-to-one call from a client troupe. The general case therefore factors into the two special cases already described; no additional algorithms are required for the general case. Each client troupe member executes the one-to-many algorithm (as if it were an unreplicated client calling the server troupe), and each server troupe member executes the many-to-one algorithm (as if it were an unreplicated server handling an incoming call from the client troupe).

Observe also that there is never any communication between members of the same troupe in the five steps listed above; communication occurs only between members of different troupes. This means that nowhere in a troupe member is there any information about other members of its own troupe, or whether it is replicated at all. Neither the protocol routines in the run-time system nor the stub procedures produced by the stub compiler use such information.

Finally, notice that messages are sent only in steps 1 and 4, and in both these steps, the message is sent to an entire troupe. Thus, call messages are sent to the entire server troupe, and **return** messages are sent to the entire client troupe. These steps obviously correspond to multicast operations.

A multicast implementation makes a dramatic difference in the efficiency of the replicated procedure call protocol. Suppose that there are m client troupe members and n server troupe members. Point-to-point communication requires a total of mn messages to be sent. In contrast, a multicast implementation requires only $m + n$ messages to be sent.

4.3.4 Waiting for Messages to Arrive

A client making a one-to-many call requires a single result, but it receives a **return** message from each server troupe member. Similarly, a server handling a many-to-one call must perform the requested procedure once, but it receives a call message from each client troupe member.

Since troupes are assumed to be deterministic, all the messages in these sets will be identical. When should computation proceed: as soon as the first message arrives, or only after the entire set has arrived?

Waiting for all messages to arrive and checking whether they are identical is analogous to providing *error detection* as well as transparent *error correction*. Any inconsistency among the messages is detected, but the execution time of the replicated program as a whole is determined by the slowest member of each troupe. This *unanimous* approach is used by default in the Circus system.

If one is willing to forfeit such error detection, then a *first-come* approach can be used, in which computation is allowed to proceed as soon as the first message in each set arrives. In this case, the execution time of the program as a whole is determined by the fastest member of each troupe.

The first-come approach requires only a simple change to the one-to-many call protocol. The client can use the call sequence number provided by the paired message protocol to discard return messages from slow server troupe members.

The many-to-one call protocol becomes more complicated; in this respect, the first-come approach destroys the symmetry between the client and server halves of the protocol. The server must be allowed to start performing a procedure as soon as the first call message from a client troupe member arrives. When a call message for the same procedure arrives from another member of that client troupe, the server cannot execute the procedure again, because that would violate the exactly-once execution property. The server must therefore retain the return message until the corresponding call messages from all other members of the client troupe have arrived. Whenever such a call message arrives, the return message is retransmitted. Execution of the procedure thus appears instantaneous to the slow client troupe members, since the return message is ready and waiting.

Note that once a client troupe member has received the results of its call, it is free to go ahead and make more calls. Therefore, as the slower members of the client troupe fall further and further behind the faster ones, the server must buffer more and more return messages. When a call message arrives from one of the slower client troupe members, the server must be able to find its earlier response from among the buffered return messages, in order to retransmit it. The call sequence number associated with each message by the paired message protocol suffices for this purpose, because of the assumption that troupes are deterministic and the argument used in Section 4.3.2.

A better first-come scheme can be implemented by buffering messages at the client rather than the server. In this case, the server broadcasts return messages to the entire client troupe in response to the first call message. A client troupe member may receive a return message for a call message that has not yet been sent; this return message must be retained until the client troupe member is ready to send the corresponding call message.

This approach is preferable to buffering messages at the server, for the following reasons:

1. the burden of buffering return messages and pairing them with the corresponding late call messages is placed on the client, rather than a shared and potentially heavily-loaded server;

2. the server can use broadcast rather than point-to-point communication; and

3. no communication is required by a slow client once it is ready to send a call message, since the corresponding return message has already arrived.

Majority voting schemes require similar buffering of return messages. Simulations and queueing models have been used to analyze the buffering requirements in this context as a function of the variation in execution rate [116].

Error detection is desirable in practice, since programmers may not be sure that their programs are deterministic. To provide error detection and still allow computation to proceed early, a *watchdog* scheme can be used. This technique requires that the computation be structured as one or more transactions (described in Chapter 5). Computation proceeds with the first message, but another thread of control (the watchdog) waits for the remaining messages and compares them with the first. If an inconsistency is detected by the watchdog, the main computation is aborted. Note that this scheme also requires buffering (in the form of transaction workspaces) to compensate for the skew in execution rates of different troupe members.

Many other schemes are possible in addition to the approaches described here. Discovering and evaluating such variations is an important area for future research.

4.3.5 Crashes and Partitions

Whenever a troupe member is waiting for one or more messages in the one-to-many and many-to-one call algorithms, the underlying message protocol uses probing and timeouts (described in Section 4.2.3) to detect crashes. This mechanism relies on network connectivity, and therefore cannot distinguish between crashes and network partitions.

Network partitions raise the possibility of different troupe members continuing to execute, each believing that the others have crashed. To prevent troupe members in different partitions from diverging, one can require that each troupe member receive a majority of the expected set of messages before computation is allowed to proceed there.

4.3.6 Collators

One way to relax the determinism requirement (at the cost of transparency) is to allow programmers to specify their own procedures for reducing a set of messages to a single message. Such procedures are called *collators*.

A collator is a function that maps a set of messages into a single result. To improve performance, it is desirable for computation to proceed as soon as enough messages have arrived for the collator to make a decision. (This is equivalent to using *lazy evaluation* [29,35] when applying the collator.) A solution to the problem of how to incorporate collators at the programming language level is presented in Section 7.4.

Three collators are supported at the replicated procedure call protocol level (viewing the contents of call and return messages as uninterpreted bits): *unanimous*, which requires all the messages to be identical and raises an exception otherwise; *majority*, which performs majority voting on the messages; and *first-come*, which accepts the first message that arrives. The framework of replicated calls and collators is sufficiently general to express weighted voting [30,79] and other replicated or broadcast-based algorithms [55,76]. Programmers can define their own application-specific collators using the mechanisms described in Section 7.4.

4.3.7 Implementation Details

Nelson argues persuasively that, in the presence of concurrency, parallel invocation semantics rather than serial are needed in order to match the semantics of the local case [73]. When incoming calls are serialized by arrival time, the possibility of deadlock is introduced. This type of deadlock does not occur when incoming calls are handled by concurrent processes.

The Circus implementation suffers from this deficiency because of the lack of multiple lightweight processes within the same address space under Berkeley 4.2BSD. A partial solution has been provided in the form of a simple process mechanism for C that supports several threads of control and provides synchronization primitives for signaling and awaiting events.

The Berkeley 4.2BSD networking primitives used by Circus do not currently allow access to the multicast capabilities of the Ethernet [22]. If this functionality were added, the operation of sending the same message to an entire troupe could be implemented by a multicast operation, and the binding agent (described in Chapter 6) could manipulate Ethernet hardware group addresses.

4.4 Performance Analysis

4.4.1 Measurements

Experiments were conducted to measure the cost of replicated procedure calls as a function of the degree of replication. The cost of a simple exchange of datagrams was also measured in order to establish a lower bound.

The experiments were run on lightly loaded University computer center machines during an inter-semester break. The distributed system consisted of six identically configured VAX[1]-11/750 systems, connected by a single 10 megabit per second Ethernet cable.

Any implementation of a paired message protocol on top of an unreliable datagram layer must perform at least the following steps in the course of a message exchange:

1. Send a datagram.

2. Receive a datagram, specifying a timeout to detect lost datagrams.

The time required to perform these operations therefore represents a lower bound for any implementation of a remote procedure call protocol using unreliable datagrams.

The client and server shown in Figure 4.5 perform the above operations using UDP datagrams under Berkeley 4.2BSD. The `sendmsg` and `recvmsg` functions are general primitives for sending and receiving datagrams. The `alarm` function uses the `setitimer` primitive to manipulate a software interval timer.

A reliable byte-stream protocol, such as TCP, is generally considered to be inferior to datagrams for the purposes of a remote procedure call implementation. The TCP-based client and server shown in Figure 4.6 are included for the purpose of comparison. Unlike the UDP client, the TCP client does not need the `alarm` function, because TCP provides reliable delivery.

The client and server shown in Figure 4.7 were used to measure the performance of Circus replicated procedure calls. Recall that the current Circus protocol is implemented entirely in user mode.

The first set of experiments measured the time per procedure call as a function of the degree of replication. The time for an exchange of UDP datagrams and the time for an exchange of messages over a TCP byte-stream are included for comparison. The results are summarized in Table 4.1. The time of day and the total user-mode and kernel-mode CPU time used by the client process were recorded before and after each replicated procedure call, using the Berkeley 4.2BSD `gettimeofday` and `getrusage` system calls. The entries in

[1]VAX is a trademark of Digital Equipment Corporation.

```
client:
    procedure
        loop
            sendmsg()
            alarm(timeout)
            recvmsg()
            alarm(0)
        end loop
    end procedure

server:
    procedure
        loop
            recvmsg()
            sendmsg()
        end loop
    end procedure
```

Figure 4.5: The UDP test client and server

```
client:
    procedure
        connect to server
        loop
            write()
            read()
        end loop
    end procedure

server:
    procedure
        accept connection from client
        loop
            read()
            write()
        end loop
    end procedure
```

Figure 4.6: The TCP test client and server

```
rpctest:
    interface
        buffer: type = array of bytes
        echo: procedure (buffer) returns (buffer)
    end interface

client:
    module
        imports rpctest
        b: buffer
        begin
            loop
                b := rpctest.echo(b)
            end loop
        end
    end module

server:
    module
        exports rpctest
        echo:
            procedure (argument: buffer) returns (result: buffer)
                result := argument
            end procedure
    end module
```

Figure 4.7: The RPC test client and server

degree of replication	real time (msecs/rpc)	total cpu time (msecs/rpc)	user cpu time (msecs/rpc)	kernel cpu time (msecs/rpc)
(UDP)	26.5	13.3	0.8	12.4
(TCP)	23.2	8.3	0.5	7.8
1	48.0	24.1	5.9	18.2
2	58.0	45.2	10.0	35.2
3	69.4	66.8	13.0	53.8
4	90.2	87.2	16.8	70.4
5	109.5	107.2	21.0	86.1

Table 4.1: Performance of UDP, TCP, and Circus

Table 4.1 were calculated by averaging the differences between the before and after values for each component of the execution time.

Note that the TCP echo test is faster than the UDP echo test. Several factors help explain this somewhat surprising result. First, the cost of TCP connection establishment is effectively ignored, since it is amortized over the `read` and `write` loop. Second, the UDP-based test makes two `alarm` calls, and therefore two `setitimer` system calls, which take approximately 1.2 milliseconds each (see Table 4.2); the corresponding TCP timers are managed by the kernel. Finally, the `read` and `write` interface to TCP byte-streams is more streamlined than the `sendmsg` and `recvmsg` interface to UDP datagrams, which uses *scatter/gather I/O*. The scatter/gather interface uses an array of address/length pairs to specify the location in user space of the datagram to be received or sent. The array is first copied from user to kernel space, and then the pieces of the datagram specified by the array are transferred between user and kernel space. This additional copying does not occur when the `read` and `write` system calls are used.

An unreplicated Circus remote procedure call requires almost twice the time of a simple UDP exchange. This is largely due to the extra system calls required to handle various aspects of the Circus protocol. The use of interrupt-driven I/O and timers, for example, requires substantial trafficking with the software interrupt facilities in order to protect critical regions. It is worth noting that these facilities are used by Circus to compensate for the lack of multiple lightweight processes within the same address space under Berkeley 4.2BSD.

Another added expense is the presence of fairly elaborate code to handle *multi-homed* machines (machines with more than one network address). In the research computer network at Berkeley, some machines have as many as four network addresses. The `sendmsg` system call does not allow a source address to be specified when the sender is multi-homed. This means that a multi-homed server is unable to ensure that its reply to a client bears the same network address that the client used in reaching the server. The only way around this problem in the current Berkeley 4.2BSD system is for a multi-homed server to use an array of sockets, one for each of its addresses, and to use the `select` system call to multiplex among them. This situation is a design oversight in Berkeley 4.2BSD, not a fundamental problem.

The incremental expense of a Circus replicated procedure call as the degree of replication increases is more reasonable. Table 4.1 shows that each additional server troupe member adds between 10 and 20 milliseconds to the real time per call. The fact that this is smaller than the time for a UDP datagram exchange shows that the replicated procedure call protocol achieves some parallelism among the message exchanges with server troupe members, but it is still the case that each component of the time per call increases linearly with the size of the troupe. This linear increase is shown in Figure 4.8.

In the second set of tests, an execution profiling tool was used to analyze the Circus implementation in finer detail. The profiles showed that six Berkeley 4.2BSD system calls account for more than half of the total CPU time used to perform a replicated procedure call. Table 4.2 shows the CPU time for each of these primitives. Table 4.3 shows the percentage of the total CPU time for a replicated call that each of these system calls accounts for, as a function of the degree of replication.

These measurements show that most of the time required for a Circus replicated procedure call is spent in the simulation of multicasting by means of successive `sendmsg` oper-

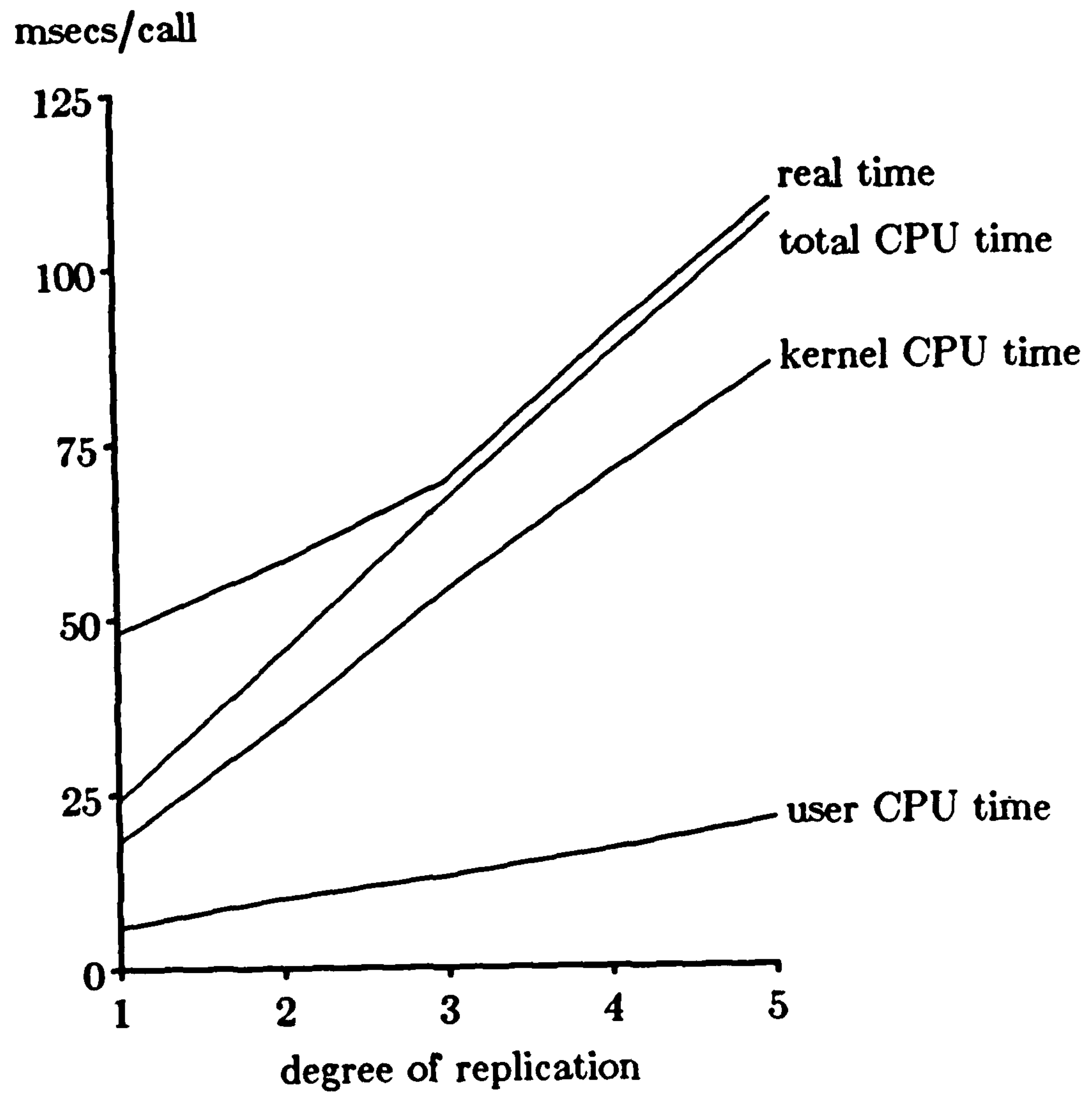

Figure 4.8: Performance of Circus replicated procedure calls

system call	msecs/call	description
sendmsg	8.1	send datagram
recvmsg	2.8	receive datagram
select	1.8	inquire if datagram has arrived
setitimer	1.2	start interval timer for clock interrupt
gettimeofday	0.7	get time of day
sigblock	0.4	mask software interrupts to begin critical region

Table 4.2: CPU time for Berkeley 4.2BSD system calls used in Circus

degree of replication	percentage of total CPU time spent in:						
	sendmsg	select	recvmsg	setitimer	gettimeofday	sigblock	total
1	27.2	11.2	9.2	4.4	2.2	1.7	55.9
2	28.8	12.7	10.6	3.5	2.7	1.2	59.5
3	32.5	11.7	11.9	4.2	2.6	1.0	63.9
4	32.9	10.3	10.7	5.4	2.9	0.8	63.0
5	33.0	9.9	11.1	5.0	3.1	0.9	63.0

Table 4.3: Execution profile for Circus replicated procedure calls

ations, and that sendmsg is the most expensive of the Berkeley 4.2BSD primitives used by the Circus implementation.

4.4.2 Theoretical Analysis

Communication dominates the time for a replicated procedure call in the Circus implementation, but that may not be true for future implementations. Another factor to be considered is the inherent delay introduced by waiting for all replicas of a procedure execution to complete. This delay might become noticeable in a more efficient, multicast-based implementation of replicated procedure calls.

Consider the following probabilistic model. A single client makes a replicated procedure call to a server troupe of size n by multicasting a single call message and receiving all n return messages. Let T_i be the time at which the client receives the return message from server troupe member i. The total time for the replicated call is the random variable $T = \max(T_1, \ldots, T_n)$. The goal of this analysis is to derive the expected value of T from the expected values of $T_1, \ldots, T_n$.

Some preliminary mathematical definitions and results are required for the remainder of the analysis.

Definition 4.1. The *harmonic numbers* H_n are defined by

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}$$

Theorem 4.2.

$$\int_0^1 nx^{n-1} \log \frac{1}{1-x}\, dx = H_n$$

Proof:

$$\begin{aligned}
\int_0^1 nx^{n-1}\log\frac{1}{1-x}\,dx &= \sum_{k=1}^{\infty}\int_0^1 \frac{nx^{n+k-1}}{k}\,dx \\
&= \sum_{k=1}^{\infty}\frac{n}{k(n+k)} \\
&= \sum_{k=1}^{n}\frac{1}{k} \\
&= H_n
\end{aligned}$$

Theorem 4.3. Let $X_1, \ldots, X_n$ be independent random variables, each exponentially distributed with mean $\mathrm{E}[X_i] = 1/\mu$, and let $X = \max(X_1, \ldots, X_n)$. Then $\mathrm{E}[X] = H_n/\mu$.

Proof: Let $F(x) = \mathrm{P}[X_i \le x] = 1 - e^{-\mu x}$ be the probability distribution function (PDF) of each X_i, and let $f(x) = dF(x)/dx = \mu e^{-\mu x}$ be the corresponding probability density function (pdf). Then the PDF of X is given by

$$\begin{aligned}
\mathrm{P}[X \le x] &= \mathrm{P}[\max(X_1, \ldots, X_n) \le x] \\
&= \mathrm{P}[X_1 \le x \wedge \ldots \wedge X_n \le x] \\
&= \mathrm{P}[X_1 \le x] \cdots \mathrm{P}[X_n \le x] \\
&= F(x)^n \\
&= (1 - e^{-\mu x})^n
\end{aligned}$$

and the pdf of X is therefore

$$\begin{aligned}
\frac{d}{dx}F(x)^n &= nF(x)^{n-1}f(x) \\
&= n(1 - e^{-\mu x})^{n-1}\mu e^{-\mu x}
\end{aligned}$$

So the expected value of X is

$$\mathrm{E}[X] = \int_0^{\infty} nx(1 - e^{-\mu x})^{n-1}\mu e^{-\mu x}\,dx$$

Substituting $t = 1 - e^{-\mu x}$, we obtain

$$\begin{aligned}
\mathrm{E}[X] &= \frac{1}{\mu}\int_0^1 nt^{n-1}\log\frac{1}{1-t}\,dt \\
&= H_n/\mu
\end{aligned}$$

by Theorem 4.2. $\square$

Returning now to the analysis of replicated procedure calls, suppose that each T_i is exponentially distributed with mean r. Then by Theorem 4.3, $\mathrm{E}[T] = H_n r$. It can be shown that $H_n = \log n + O(1)$ (see Knuth [46], for example). Therefore,

$$\mathrm{E}[T] = r\log n + O(r)$$

This equation provides an estimate of the average time for a replicated procedure call, given an efficient multicast implementation and exponentially distributed round-trip times. The expected time per call increases only logarithmically with the size of the troupe.

Compare this with an implementation (like Circus) that simulates multicast operations by repeated point-to-point transmissions. Table 4.2 shows that each sendmsg operation takes 8 milliseconds. Since the sendmsg operation is two orders of magnitude slower than either the time to transmit a packet over the network or the time to execute a simple procedure on a remote machine, the time per call increases linearly with the size of the troupe. This conclusion is borne out by the measurements shown in Figure 4.8.

The foregoing analysis emphasizes the importance of efficient multicast operations in an implementation of replicated procedure call.

Chapter 5

Replicated Transactions

This chapter discusses the problem of synchronizing concurrent threads of control within a replicated distributed program. Transactions are adopted because they provide both the necessary synchronization and a convenient way to undo partially completed computations.

A troupe commit protocol is presented. The protocol guarantees that all troupe members commit transactions in the same order, but requires no communication among troupe members. The protocol is *generic* because any local concurrency control method can be used by any individual troupe member, as long as it correctly serializes the effects of transactions. The protocol is *optimistic* because it assumes that concurrent transactions are unlikely to conflict.

A probabilistic model is used to analyze the performance of the troupe commit protocol. The analysis shows that it is subject to starvation when many concurrent transactions conflict.

An alternative starvation-free scheme is presented for situations in which the troupe commit protocol proves unacceptable. This method, based on an ordered broadcast protocol, does not introduce starvation, but restricts the local concurrency control algorithms that can be used at each troupe. The local concurrency control algorithms must commit transactions in an order that is a well-defined function of their arrival order.

5.1 The Synchronization Problem for Troupes

Multiple threads of control give rise to concurrent calls from different client troupes to the same server troupe. This is not the same as a many-to-one call, which is handled by the algorithms described in Section 4.3. For a module to operate correctly in the presence of concurrent calls from different clients, even without replication, it must appear to execute those calls in some serial order. Serializability can be achieved by any of a number of concurrency control algorithms [4,47,59].

When the server module is a troupe, not only must concurrent calls from different client troupes be serialized by each server troupe member, but they must be serialized in the *same order*. Le Lann [59] describes this synchronization requirement as follows:

> Fully replicated computing refers to a situation where every action fired by any operation must be processed by all consumers. ...In this context, the purpose of a synchronization mechanism is to guarantee that the ordering of actions processed by consumers is identical for all consumers.

Correct semantics require proper coordination between the replicated procedure call mechanism and a synchronization mechanism, such as nested atomic actions [70,71,84,85]. This chapter presents the design of such a unified mechanism, which also allows the determinism constraints on troupe members to be loosened.

5.2 Replicated Lightweight Transactions

A transaction is expressed as a sequence of programming language statements bracketed by **transaction** and **end transaction** keywords. Transactions can be dynamically nested, just like procedure activation records. A transaction commits when it reaches **end transaction** or when it returns normally. A transaction aborts when an explicit **abort** statement is reached or when an exception is raised.

The transaction mechanism for troupes must guarantee serializability and atomicity, so that when a transaction aborts, its tentative updates and the committed updates of its subtransactions can be undone without affecting other concurrently executing transactions.

The nested transaction mechanisms for conventional programs described by Reed [84,85] and Moss [70,71] not only provide these two properties, but they also guarantee the *permanence* of committed updates. Stable storage is used for intention lists and commit records, and the commit algorithm is coupled with a crash recovery algorithm.

This third property (permanence) is not required in programs constructed from troupes, because troupes automatically mask partial failures. Consequently, an implementation of transactions for replicated distributed programs can dispense with the crash recovery facilities based on stable storage and operate entirely in volatile memory. The result is a more efficient form of transactions called *lightweight transactions.*

5.2.1 Correctness Condition

The correctness condition for conventional transactions is serializability. With troupes, however, independent serialization of transactions at each troupe member is not enough: troupe consistency must also be preserved.

A sufficient condition for preserving troupe consistency is to ensure that all troupe members serialize transactions in the same order. Existing concurrency control algorithms for replicated databases guarantee identical serialization orders at all replicas, but many of these algorithms require communication among replicas [4,59]. The desire for troupe members to remain unaware of one another's existence rules out the use of such algorithms.

One well-known multiple-copy concurrency control algorithm requires no inter-replica communication: two-phase locking with unanimous update [4]. This algorithm requires each replica to use two-phase locking for local concurrency control. The protocol presented in the next section removes this restriction.

5.3 A Troupe Commit Protocol

The protocol described in this section is *optimistic*, because it assumes that concurrent transactions are unlikely to conflict, and it is *generic*, because it assumes nothing about the local concurrency control algorithms used by the individual troupe members. The protocol detects any attempt by troupe members to serialize transactions differently, and transforms such an attempt into a deadlock. Deadlock detection is then used to abort and retry one or more of the offending transactions [31,75].

When a server troupe member is ready to commit or abort a transaction, it calls the following `ready_to_commit` procedure.

```
ready_to_commit: procedure (boolean) returns (boolean)
```

A **true** argument means that the server troupe member is ready to commit the transaction; a **false** argument means that the server troupe member wishes to abort the transaction. If the `ready_to_commit` procedure returns **true**, the server troupe member goes ahead and commits the transaction; otherwise, the transaction is aborted. The server's call is translated into a remote call to the client troupe. The roles of client and server are thus temporarily reversed; this is known as a *call-back protocol*.

Each member of the client troupe implements the `ready_to_commit` procedure as follows. If each server troupe member calls `ready_to_commit(true)`, then the client returns **true** to the entire server troupe, and the transaction is committed at each server troupe member. If any server troupe member calls `ready_to_commit(false)`, then the client returns **false** to the entire server troupe, and the transaction is aborted at each server troupe member. Each member of the client troupe thus plays the role of the coordinator in the conventional two-phase commit protocol [31,53,57].

The troupe commit protocol has the following essential property.

Theorem 5.1. Two troupe members succeed in committing two transactions if and only if both troupe members attempt to commit the transactions in the same order.

Proof: Let S_1 and S_2 be two members of a server troupe S, and let C and C' be two client troupes. Suppose C performs transaction T at S, and C' performs transaction T' at S.

If both S_1 and S_2 commit T and T' in the same order, say T followed by T', then both call `ready_to_commit` first at C and then at C'. Client C tells both troupe members to go ahead, C' does the same, and both S_1 and S_2 succeed in committing the transactions.

Now suppose that S_1 tries to commit T first, but S_2 tries to commit T' first. Then S_1 calls `ready_to_commit` at C and S_2 calls `ready_to_commit` at C'. The result is deadlock, because the `ready_to_commit` procedure at each client waits for all members of the server troupe to become ready before responding to any of them. Therefore, neither S_1 nor S_2 succeeds in committing either transaction. □

The troupe commit protocol therefore ensures that all troupe members commit transactions in the same order.

Note that it is only necessary to transform different serialization orders into deadlocks when the different serialization orders would cause inconsistent states at troupe members. If the transactions being serialized do not conflict with one another, then inconsistency cannot occur, yet the protocol above may still cause deadlock. To remedy this, the local concurrency control algorithm should commit non-conflicting transactions in parallel. For example, using the notation from the proof of Theorem 5.1, suppose that transactions T and T' do not conflict. Then S_1 and S_2 commit T and T' in parallel, so S_1 and S_2 call both `ready_to_commit` at C and `ready_to_commit` at C' in parallel. The deadlock described above does not occur, because the `ready_to_commit` procedure at each client receives calls from both S_1 and S_2.

5.3.1 Performance Analysis

Suppose a server troupe has n members, and suppose that the number of conflicting transactions at some moment is $k \geq 1$. (Transactions that do not conflict are assumed to be committed in parallel, as discussed above.) Each member of the server troupe serializes these k transactions in some order. To be conservative, assume that each of the $k!$ possible orderings is equally likely at each troupe member, and that the troupe members perform their serializations independently. The troupe commit protocol of Section 5.3 is free of deadlock when all members of the server troupe serialize all transactions in the same order. The probability that the n troupe members independently choose the same serialization order is $(1/k!)^{n-1}$, so the probability of deadlock introduced by the troupe commit protocol is

$$\mathrm{P}[\text{deadlock}] = 1 - \left(\frac{1}{k!}\right)^{n-1} \tag{5.1}$$

The above analysis shows that the probability of deadlock rapidly approaches certainty when the optimistic assumption of few conflicting transactions fails to hold. The troupe commit protocol is therefore subject to *starvation*. One way to alleviate the problem is to use binary exponential back-off [67] when retrying transactions aborted after a deadlock. In this scheme, an aborted transaction is delayed for a randomly chosen interval before being retried. If successive retries are required, the mean delay is doubled each time.

5.4 A Starvation-Free Algorithm

The algorithm described in this section is *starvation-free*, because it does not introduce any additional chance of deadlock (unlike the troupe commit protocol). It uses an *ordered broadcast* protocol and requires a *deterministic local concurrency control* algorithm at each troupe member.

The ordered broadcast protocol guarantees that concurrent broadcasts are never interleaved: all recipients of broadcast messages accept them for application-level processing in the same order. The protocol assumes synchronized clocks [50]. It is a modification of an atomic broadcast algorithm due to Skeen [95], which also handles failures of the sender and recipients. The ordered broadcast protocol is simpler because the replicated structure of troupes obviates the need for this type of crash recovery.

The ordered broadcast protocol is shown in Figure 5.1. The protocol is expressed in terms of replicated procedure calls, using the explicit replication approach of Section 7.4. The `P(x) at troupe` construct indicates a replicated call to the specified troupe.

The protocol involves two phases, based on the procedures `get_proposed_time` and `accept_time`. A client wishing to perform an ordered broadcast begins by calling `get_proposed_time` with its message. Each server troupe member responds to the `get_proposed_time` procedure by inserting the message in its queue and proposing a time at which it will accept the message fo: processing. The client then calls `accept_time` with the maximum of these proposed times. Each server troupe member performs the `accept_time` procedure by changing the status of the message to `accepted` and changing the position of the message in the queue to reflect the accepted time for it. A server only accepts a message for processing if its status is `accepted`, its acceptance time has actually arrived, and there are no messages with earlier proposed times that have not yet been accepted. The members of the server troupe therefore accept these broadcast messages in the same order.

The starvation-free concurrency control scheme requires that replicated transactions be initiated by means of the ordered broadcast algorithm, so that each transaction has the

```
-- Client side
atomic_broadcast:
    procedure (message, troupe)
        proposals := get_proposed_time(message) at troupe
        max := 0
        for time in proposals() do
            if time > max then max := time
        end for
        accept_time(message, max) at troupe
    end procedure

-- Server side
status: type = {proposed, accepted}
message_queue: queue of (message, time, status) -- ordered by time

get_proposed_time:
    procedure (message) returns (time)
        time := now()  -- current time from synchronized clock
        insert (message, time, proposed) into message_queue
        return time
    end procedure

accept_time:
    procedure (message, accepted_time)
        remove (message, -, proposed) from message_queue
        insert (message, accepted_time, accepted) into message_queue
        loop
            (message, time, status) := head of message_queue
            if status = proposed or time > now() then
                exit
            end if
            -- accept message for application-level processing
        end loop
    end procedure
```

Figure 5.1: The ordered broadcast protocol

same time stamp (its acceptance time) at each troupe member. Furthermore, it requires the same deterministic concurrency control algorithm at each troupe member.

A concurrency control algorithm is deterministic if it guarantees that the serialization order of a set of concurrent transactions is a well-defined (deterministic) function of the order in which they arrived. When combined with ordered broadcast, the deterministic concurrency control algorithm ensures that all troupe members serialize transactions in the same order.

A trivial example of a deterministic concurrency control algorithm is serial execution in chronological order, but the lack of concurrency makes it unacceptable. Another possibility is the combination of time stamps and two-phase locking described by Rosenkrantz *et al.* [86]

5.5 Discussion

Both of the approaches to synchronization presented in this chapter have disadvantages. The troupe commit protocol of Section 5.3 is subject to starvation under conditions of heavy load. The starvation-free scheme of Section 5.4 limits the potential concurrency that can be achieved.

An alternative is to take a programming-in-the-large approach to synchronization, in which the choice of concurrency control scheme is made on a module-by-module basis. Most modules in a replicated distributed program could use the optimistic commit protocol, while those modules subject to high degrees of concurrency could use the ordered broadcast protocol.

Attempting to provide general-purpose concurrency control with no knowledge of the semantics of the application may be too ambitious. Application-specific synchronization will occasionally be required. The semantics of the Grapevine system, for example, make eventual convergence an acceptable alternative to instantaneous consistency.

Chapter 6

Binding and Reconfiguration

This chapter reviews the binding problem for unreplicated distributed programs, and then addresses the additional problems of binding and reconfiguring programs constructed from troupes. Ringmaster, the binding agent for the Circus system, is described. The mechanics of adding a new troupe member to an existing troupe are discussed. A probabilistic analysis of crash and replacement of troupe members is used to express the availability of a troupe as a whole in terms of the lifetime, replacement time, and degree of replication of its members. The question of how long to wait before replacing defunct troupe members is also answered.

6.1 Binding Agents for Distributed Programs

A binding agent is a mechanism that enables programs to import and export modules by interface name. In the case of distributed programs constructed with remote procedure calls, the interface name must be associated with the address of the server that exports it, and must be looked up by the client that imports it. These functions (registration, lookup, and perhaps deletion) can be provided by a general-purpose name server. For example, Grapevine [9] is used as the binding agent in the Xerox PARC RPC system [10], and Clearinghouse [76] plays the same role for Courier [115].

A natural means of reducing the cost of name server lookups is to have clients cache the results of such lookups. Thus, a client contacts the binding agent only when it imports an interface, and it uses the same information for all subsequent remote calls to that module. This raises the classic *cache invalidation problem*: what happens when a client's binding information becomes stale because the information at the name server has changed?

Suppose a client makes a remote call to a server using its cached information. In the case of programs constructed from conventional remote procedure calls, there are three reasons why the cached information might be stale:

1. There is no longer a server at the specified address.
2. There is a server at that address, but it no longer exports the specified interface.
3. There is a server at that address and it exports that interface, but the actual instance of the module in question is no longer the same as the one originally imported by the client.

If all three of these cases can be detected at or below the remote procedure call protocol level, the run-time system can raise an exception in the client to indicate that rebinding *is required. Therefore, the problem of masking stale* binding information reduces to the problem of detecting the above three cases.

The first can be detected at the paired message protocol level, since there will be no response to repeated retransmissions. The second can be detected at the remote procedure call protocol level, because the server's run-time system will reject the call. The third case requires some help from the binding agent, in the form of *incarnation numbers* for exported interfaces, as in the system described by Birrell and Nelson [10]. This scheme time-stamps the record created when a server registers itself with the binding agent. The client's run-time system receives the time stamp along with the server address when it imports an interface and includes it in all subsequent calls to that module. It is thus a simple matter for the server's run-time system to detect and reject mismatches.

A related problem is *garbage collection*, which is required when some of the binding agent's own registration information becomes obsolete. This can happen if a server crashes or otherwise ceases to export an interface without informing the binding agent. The problem of garbage collection reduces to the cache invalidation problem, since the information maintained by the binding agent is itself just a cached version of the truth. Of the above three ways in which binding information can be out of date, only the first two apply to the binding agent. The third case is detected by the binding agent itself as part of the process of assigning incarnation numbers: when a server re-exports an interface, the binding agent will notice that there is already an entry for that name and address. In the first two cases, however, it is the client that ends up detecting the invalid binding; this fact must somehow reach the binding agent.

One solution is to include a special `rebind` procedure in the interface to the binding agent. Each client, upon detecting an invalid binding, calls `rebind` with the invalid binding as an argument. The binding agent looks up and returns the current binding for the given name, and deletes the old binding if it is still present. (The old binding passed to the `rebind` procedure is only a hint; it need not be deleted immediately, nor should it be blindly accepted as invalid in an insecure environment.)

Another solution is to use a garbage collector: a process which periodically enumerates all the registered modules, probes them with a special null procedure call (an "are you there?" request), and explicitly deletes the bindings for modules that do not respond. The garbage collector need not be part of the binding agent if the binding interface includes enumeration and deletion.

6.2 Binding Agents for Replicated Programs

Replicated distributed programs import and export troupes rather than single modules, and therefore require additional support from the binding mechanism. First of all, the binding agent must manipulate sets of module addresses rather than single addresses, and it must manage the troupe IDs required by the replicated procedure call algorithms of Section 4.3. The binding agent must allow a third party (such as the configuration manager of Section 7.5.3) to register an entire troupe. Finally, it must be possible to add or delete individual troupe members, in order to handle troupe reconfiguration.

Since binding is such a pivotal mechanism, it is essential that the binding agent be highly available. An obvious choice is to make the binding agent a troupe and express the interactions with it in terms of replicated procedure calls. The interface to such a binding agent is shown in Figure 6.1.

The initial registration of a troupe also requires the ability to add a member to an existing troupe. A troupe cannot register itself *en masse* with a single replicated procedure call, because it does not have a troupe ID until it is registered. To avoid this

```
binding:
    interface
        troupe_name: type = string
        troupe_member: type = module_address
        troupe: type = set of troupe_member
        troupe_id: type = unique_id

        register_troupe:
            procedure (troupe_name, troupe) returns (troupe_id)
        add_troupe_member:
            procedure (troupe_name, troupe_member) returns (troupe_id)
        lookup_troupe_by_name:
            procedure (troupe_name) returns (troupe)
        lookup_troupe_by_id:
            procedure (troupe_id) returns (troupe)
    end interface
```

Figure 6.1: The interface to the binding agent

circularity, each troupe member must add itself individually to an initially empty troupe, using add_troupe_member. The synchronization requirements of the add_troupe_member operation are discussed below.

The cache invalidation problem becomes more complicated when replication is introduced. Let T be the set of members of a troupe, and let C be the cached set of members that a client believes constitutes the troupe. Then C is stale if and only if $C \neq T$. The possibilities for stale information correspond to the possible intersections of these two nonempty sets:

1. $T \cap C = \emptyset$
2. $T \subset C$
3. $T \supset C$
4. $T \cap C \neq \emptyset \wedge T \not\subset C \wedge T \not\supset C$

The semantics of troupes and replicated procedure calls require every member of a server troupe to execute a procedure if any member does. This will be the case if $T = C$, $T \cap C = \emptyset$, or $T \subset C$. The first two possibilities for stale information are therefore harmless; the client will detect that some or all of the members of C are invalid, and perform the necessary rebinding. In the last two cases, however, the client calls some but not all of the troupe members; these calls cannot be allowed to succeed.

The solution is to use troupe IDs as a form of incarnation number. Each call message carries the troupe ID of its destination as well as its source, and each server troupe member rejects any call message whose destination troupe ID is incorrect. If it can be guaranteed that a troupe always changes both its membership and its troupe ID in an atomic operation, then the problem is solved: a server troupe member accepts a call from a client only if it

```
add_troupe_member:
    procedure (troupe_name, troupe_member) returns (troupe_id)
        transaction
            troupe := lookup_troupe_by_name(troupe_name)
            troupe := union(troupe, {troupe_member})
            troupe_id := new_troupe_id()
            set_troupe_id(troupe_id) at troupe
            return troupe_id
        end transaction
    end procedure
```

Figure 6.2: Adding another member to a troupe

bears the correct server troupe ID, which is the case only if the client knows the correct membership of that server troupe.

The `add_troupe_member` procedure must therefore be an atomic transaction that also changes the troupe ID. This requires informing the existing members of the troupe that their troupe ID has changed, which can be accomplished by running a special `set_troupe_id` procedure at each member. The `set_troupe_id` procedure for each troupe can be generated automatically, in the same way that stub procedures are produced (see Section 7.1). If `set_troupe_id` is executed as a subtransaction of `add_troupe_member`, the change in troupe ID and troupe membership will happen atomically and will be correctly serialized with any other calls to the server troupe.

Code for the `add_troupe_member` procedure is shown in Figure 6.2. Note that both the addition of the new member and the change in troupe ID occur within an atomic transaction. The `set_troupe_id(troupe_id) at troupe` construct indicates a replicated call to the `set_troupe_id` procedure, because all members of `troupe` must be informed of the new troupe ID.

6.3 The Ringmaster Binding Agent

The Ringmaster is the binding agent for troupes in the Circus system. It is a specialized name server that enables programs to import and export troupes by name. It plays the same role that Grapevine does in the Xerox PARC RPC system [9,10]. The main differences are that the Ringmaster

- manipulates troupes (sets of module addresses),
- is a dedicated binding agent, and
- is itself a troupe whose procedures are invoked via replicated procedure calls.

The Ringmaster and its clients make use of the following types of objects:

module names

A module name is what a program uses to import or export a module, and is determined by the programming environment. Module names are represented as character strings.

module addresses

A module address (defined in Section 4.3) uniquely identifies an instance of a module in the internet.

troupes

A troupe is represented by the set of module addresses of its members.

troupe IDs

A troupe ID corresponds to a unique troupe in the internet. Since troupes may be long-lived, a permanently unique ID is used.

The interface to the Ringmaster is essentially identical to the binding interface shown in Figure 6.1.

A client imports a module by calling `lookup_troupe_by_name`. This procedure returns the set of module addresses associated with that name.

A server exports a module by calling `add_troupe_member`. If there is already a troupe associated with the specified name, the exported module is added to it as a member; otherwise, a new troupe is created with the exported module as its only member. The troupe ID is returned.

The UNIX process ID of the server process is also recorded in the entry for the module, so that the Ringmaster can periodically perform garbage collection of troupe members whose processes have terminated.

A server handling a many-to-one call uses `lookup_troupe_by_id` to map the client troupe ID into the set of module addresses of the members of the client troupe (as described in Section 4.3).

Access to the binding procedures is by means of stubs produced by the stub compiler from the Ringmaster interface. These stubs are part of the Circus run-time system.

Since the Ringmaster cannot be used to import itself, a special degenerate binding mechanism is used for the Ringmaster module: the Ringmaster troupe is partially specified by means of a well-known port on each machine, and the set of machines running instances of the Ringmaster is determined at run-time. Currently, a configuration file is used for this purpose; a better solution would be a broadcast protocol.

6.4 Reconfiguration and Recovery from Partial Failures

A troupe is resilient to *partial failures*, in which at least one of its members continues to function. Machine crashes are detected (using a timeout as described in Section 4.2.3) by the paired message protocol, which raises an exception that can be used by higher level software. At some point it becomes desirable to replace troupe members that have crashed, because a diminished troupe is more vulnerable to future crashes.

6.4.1 Adding a New Troupe Member

Adding a new troupe member to an existing troupe requires the following two steps:

1. the new member must be brought into a state consistent with that of the other members, and

2. the new member must be registered with the binding agent.

This section describes how to perform the first step; the add_troupe_member procedure (described in Section 6.2) is used to accomplish the second step.

The solution is to use a mechanism similar to checkpointing. In this scheme, the state information of an existing troupe member is externalized (converted to a standard external representation), then transmitted to the newly created troupe member, where it is internalized. The transmission method for abstract data types proposed by Herlihy and Liskov is similar [36].

A special get_state procedure can be produced automatically by a stub compiler (discussed in Section 7.1) for this purpose. The get_state procedure copies the module state from the callee to the caller and handles the details of externalization and internalization. This procedure executes as a read-only atomic transaction, so that the state cannot be affected while a new troupe member is being initialized. A new server process wishing to join a troupe initializes its state by making a replicated call to the get_state procedure at the existing members of the troupe, and then calls the binding agent's add_troupe_member procedure to register itself. Since the states of the existing troupe members are consistent, and since get_state is free of side effects at the callee, the replicated call to get_state is not strictly necessary; an unreplicated call to any of the existing troupe members would suffice.

Finally, the call to add_troupe_member and the call to get_state must be bracketed together in a single atomic transaction, to guarantee that the new member joins the troupe and acquires the correct state as an indivisible operation.

6.4.2 Analysis of the Reliability of Troupes

The availability A of a troupe is defined to be the equilibrium probability that the troupe is functioning. The goal of this analysis is to relate A to the average lifetime of individual troupe members and the average time to replace a failed troupe member.

Suppose a troupe has n members. Call a crash of a troupe member a failure, and call its replacement a repair. For each functioning troupe member, the lifetime, or time until failure, is assumed to be exponentially distributed with mean $1/\lambda$, and for each failed troupe member, the repair time is assumed to be exponentially distributed with mean $1/\mu$. Equivalently, each troupe member has an average failure rate of λ and an average repair rate of μ. Troupe members are assumed to fail and be repaired independently of one another.

With these assumptions, a troupe can be modeled as a special kind of discrete-state, continuous-time Markov process, called a *birth-death process*, in which transitions occur only between adjacent states. The birth-death model of a troupe is shown in Figure 6.3. The $n+1$ possible states of the system represent the number of troupe members that have failed, from 0 to n. A transition from state k to state $k+1$ represents a failure; a transition from state k to state $k-1$ represents a repair. The equilibrium probability that the system is in state k is denoted p_k. Since the troupe continues to function as long as not all n members have failed, the availability A of the troupe is the probability that the system is not in state n, so $A = 1 - p_n$.

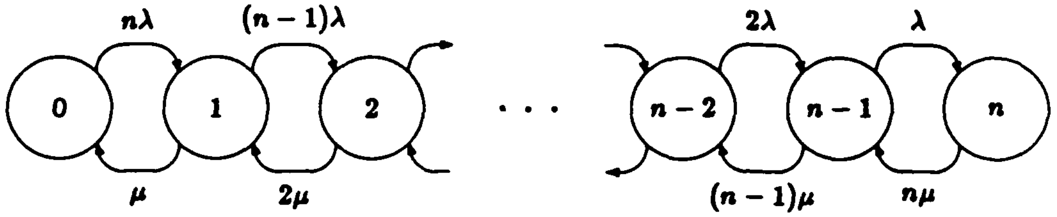

Figure 6.3: Birth-death model of troupe reliability

This birth-death process is isomorphic to a queueing system with a finite population of n customers and a server for every customer: the M/M/n/n/n queue analyzed by Kleinrock [45]. Under this isomorphism, an arriving customer corresponds to the failure of a troupe member, and a departing customer corresponds to the repair of a troupe member. Since the customer population is finite and every customer has a server, there is never more than one customer at a server. This corresponds to the fact that once a troupe member fails, it cannot fail again until after it has been repaired.

Kleinrock's analysis of the M/M/n/n/n queueing system [45] shows that p_k, the probability of k failed troupe members, is as follows.

$$p_k = \frac{\binom{n}{k}\left(\frac{\lambda}{\mu}\right)^k}{(1+\lambda/\mu)^n} \qquad k = 0, \ldots, n$$

Therefore,

$$A = 1 - p_n = 1 - \left(\frac{\lambda}{\lambda+\mu}\right)^n \tag{6.1}$$

and solving for the average replacement time $1/\mu$,

$$\frac{1}{\mu} = \frac{1}{\lambda} \cdot \frac{(1-A)^{1/n}}{1-(1-A)^{1/n}} \tag{6.2}$$

Equation 6.1 shows that, as expected, the reliability of a troupe increases dramatically when the number of troupe members increases or when the failure rate for troupe members decreases relative to the repair rate. Furthermore, given a desired degree of availability and an average lifetime $1/\lambda$ for troupe members, Equation 6.2 can be used to determine the necessary replacement time.

Suppose, for example, that a troupe consisting of three members must be available 99.9 percent of the time. Then $A = 0.999$, $1 - A = 0.001$, and $(1 - A)^{1/3} = 0.1$, so by Equation 6.2, the replacement time can be at most 1/9 of the lifetime. If each troupe member has an average lifetime of one hour, say, then the average replacement time must be no longer than 6 minutes and 40 seconds in order to achieve 99.9 percent availability.

A given level of reliability can be achieved with a longer average replacement time if the degree of replication is increased. If the troupe in the above example consisted of five members, the replacement time could be 20 minutes (1/3 of the average lifetime) and the troupe would still achieve 99.9 percent availability.

Chapter 7

Programming Language Issues

This chapter discusses some of the issues that arise when mechanisms for replicated distributed programs are integrated into programming environments. The *stub compiler* approach is used. After an overview of the stub compilation process, four examples are presented.

The first is the stub compiler used in the Circus system [19]. It translates interfaces specified in the Courier language [115] into client and server stub routines in C [44]. The stub routines take responsibility for sending parameters and results between client and server troupe members via a replicated procedure call run-time system. Two examples of adding remote procedure call to Lisp [28,103] are presented, and a stub compiler for Modula-2 [113] is described. Some observations on the relationship between programming languages and remote procedure call are presented.

Stub compilers normally hide details of the underlying system from the programmer, but there are cases in which the programmer desires explicit access to these aspects of the system. Two such cases are examined: explicit binding and explicit replication.

Finally, a configuration language and configuration manager are presented. These tools help cope with the programming-in-the-large aspects of constructing programs from troupes.

7.1 Stub Compilers

The purpose of a stub compiler is to translate a module interface into *stub procedures* for the client and server halves of a remote interface. The stub procedures are responsible for:

- communicating with the binding agent
- *externalizing* and *internalizing* objects of various data types
- passing parameters, results, and exceptions between machines

Externalization is the process of translating an object from its internal form to an external representation as a sequence of bytes, and internalization is the inverse process of translating the sequence of bytes back into the internal representation of the object; see Figure 7.1. Nelson calls these two translation processes *marshaling* and *unmarshaling* [73].

The stub procedures typically require a *run-time system* that provides access to the transport protocol and binding agent. The run-time system is also responsible for providing correct procedure invocation semantics, by creating and destroying server processes as appropriate. In Circus, for example, the run-time system implements the paired message and replicated procedure call protocols, the top-level server loop that accepts incoming calls, and the import and export procedures for contacting the Ringmaster binding agent.

Table 7.1 lists four stub compilers constructed during the course of this research, and Table 7.2 characterizes the programming languages used in these stub compilers.

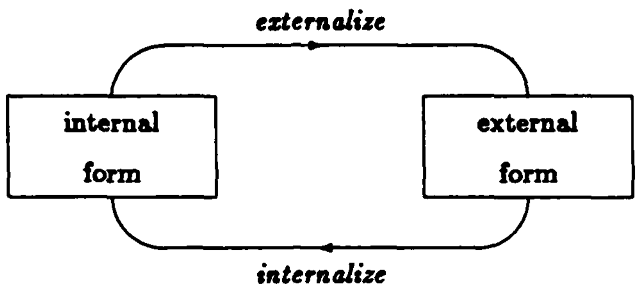

Figure 7.1: Externalization and internalization

interface language	stub language	
Courier	C	(compiled)
Courier	Lisp	(interpreted)
Lisp	Lisp	(interpreted)
Modula-2	Modula-2	(compiled)

Table 7.1: Stub compilers

programming language	type declarations	compile-time checking	run-time checking
C	yes	some	no
Lisp	no	no	yes
Courier	yes	—	—
Modula-2	yes	yes	yes

Table 7.2: Summary of interface and stub languages

```
NameServer: PROGRAM 26 VERSION 1 =
 BEGIN
   -- Types.
   Name: TYPE = STRING;
   Property: TYPE = RECORD [name: Name, value: SEQUENCE OF UNSPECIFIED];
   Properties: TYPE = SEQUENCE OF Property;
   -- Errors.
   AlreadyExists: ERROR = 0;
   NotFound: ERROR = 1;
   -- Procedures.
   Register: PROCEDURE [name: Name, properties: Properties]
             REPORTS [AlreadyExists] = 0;
   Lookup: PROCEDURE [name: Name]
           RETURNS [properties: Properties]
           REPORTS [NotFound] = 1;
   Delete: PROCEDURE [name: Name]
           REPORTS [NotFound] = 2;
 END.
```

Figure 7.2: Courier specification of a remote interface

7.1.1 Courier to C

An interface specification in Courier consists of declarations of types, constants, and procedures. Some Courier features are not present in many programming languages: error types (exceptions) that procedures may report in lieu of returning a result, constants of arbitrary constructed types, and procedures that return multiple results. In particular, because of the lack of these features in C, they are not supported in this implementation.

The predefined types include Booleans, 16-bit and 32-bit signed and unsigned integers, and character strings. The constructed types are enumerations, arrays, records, variable-length sequences, and discriminated unions. An example of a Courier program is shown in Figure 7.2.

The predefined types and the enumeration, array, and record types have obvious C counterparts. The variable-length sequences and discriminated unions pose some problems when they are mapped into C, because an object of one of these types must contain run-time information (the length of the sequence or which member of the union is present) that is implicit in the Courier type, but must be made explicit in C. Furthermore, the C programmer must bear the responsibility of keeping this information consistent when these objects are manipulated by functions other than those produced automatically by the stub compiler.

The Courier protocol specifies how objects of each type are represented when transmitted in call and return messages. Most of the work of the stub routines consists of translating parameters and results between their external and internal representations. This may involve byte-swapping of integers, realignment of record fields, and storage allocation for objects of variable-length types.

```
(PROGRAM NameServer (26 1)
  TYPES (
    (Name STRING)
    (Property (RECORD (name Name) (value (SEQUENCE UNSPECIFIED))))
    (Properties (SEQUENCE Property))
  )
  ERRORS (
    (AlreadyExists 0)
    (NotFound 1)
  )
  PROCEDURES (
    (Register ARGS (Name Properties) RESULTS () ERRORS (AlreadyExists) 0)
    (Lookup ARGS (Name) RESULTS (Properties) ERRORS (NotFound) 1)
    (Delete ARGS (Name) RESULTS () ERRORS (NotFound) 2)
  )
)
```

Figure 7.3: Representation of a Courier specification in Interlisp-D

The stub compiler also produces binding stubs to import and export the interfaces that it processes. These routines make replicated procedure calls to the Ringmaster as described in Section 6.3. The representations of troupes that are returned by these binding procedures are used by the client and server stub routines. In this way, once a program has been compiled, no editing or recompilation is required to change the number or location of troupe members.

7.1.2 Courier to Lisp

Xerox Interlisp-D workstations use the Courier remote procedure call protocol to make use of services such as name lookup, printing, and filing. These services are specified in the Courier language, but are called directly from Lisp.

Remote procedure calls were added to Interlisp-D by representing each Courier specification in list structure, and using this specification to translate between the Lisp and Courier representations of data at run-time. Figure 7.3 shows the Lisp representation of the Courier program of Figure 7.2.

The Courier protocol uses the Sequenced Packet Protocol (SPP) as its underlying message transport mechanism [114,115]. SPP is integrated into Interlisp-D as a subclass of I/O streams. The functions in Figure 7.4 are the basis for Courier support in Interlisp-D. The `CourierOpen` function opens an SPP stream, performs the initial exchange of version numbers required by the protocol, and returns a stream that can be used for subsequent Courier calls to that machine. The `CourierCall` function performs a remote procedure call. It calls `CourierWrite` and `CourierRead` to externalize and internalize Lisp values, using the representations specified by the Courier protocol as the external form. These two functions take as arguments the type of the object being converted and the name of the Courier program in which that type is declared.

```
(CourierOpen machine-name) => stream
(CourierCall stream program-name procedure-name arg1 ... argN) => results
(CourierWrite stream program-name type value)
(CourierRead stream program-name type) => value
```

Figure 7.4: Interlisp-D functions for Courier remote procedure calls

7.1.3 Lisp to Lisp

Stub procedures are effectively unnecessary in pure Lisp, because the language itself defines a standard external form: the usual parenthesized representation of list structure. Externalization and internalization are trivial, thanks to the standard Lisp functions `print` and `read`. The essential property of these functions is that the result of reading a printed form is `equal` (in Lisp) to the original form.

Remote procedure calls were added to Lisp [28] using variants of the standard `print` and `read` functions: `vector-print` and `vector-read`. The `vector-print` function converts a Lisp form to a vector of bytes that can be transmitted in a message, and the `vector-read` function converts the vector of bytes back into the original Lisp form.

No attempt was made to handle objects not present in pure Lisp, such as circular or shared list structure.

7.1.4 Modula-2 to Modula-2

During the course of this research, a stub compiler was constructed at the DEC Systems Research Center for an extended version of Modula-2 [87,113]. Modula-2 is a strongly typed programming language that supports separate *definition modules* and *implementation modules*. The extended language uses a descendant of Powell's Modula-2 compiler [83]. This compiler does not compile definition modules into symbol tables for later use; instead, the `import` statement is implemented by textual inclusion. Because of this, there are only two reasonable alternatives for constructing a remote procedure call stub compiler:

1. Modify the normal compiler to produce symbol tables for definition modules, and then use these symbol tables as the input to the stub compiler.

2. Use the front end of the normal compiler, but replace the code generation phase by stub generation.

The second alternative was chosen. Once some hooks were added to the front end, the differences between the normal compiler and the stub compiler were the following:

- The initial production of the YACC [41] grammar for the language was changed from a compilation unit to a definition module.

- The code generation phase was entirely replaced by stub generation, using a different implementation of the same interface.

This approach makes it simple to keep the stub compiler in step with the normal compiler when changes are made to the language.

The input to the stub generation phase is a parse tree whose nodes represent the declarations in the definition module. This is another advantage of sharing the front end with the normal compiler: the internal form of the parse tree and the routines for manipulating it are already available.

The compiler assumes that a file named `Test.def` contains a definition module named `Test`. The stub compiler normally produces the following four files from the input file `Test.def`.

`TestRPC.def`
: The definition module for an auxiliary interface, `TestRPC`, that declares the RPC import and export procedures that clients and servers must call.

`TestServer.mod`
: The server implementation of the `TestRPC` interface. It contains the export procedure, a dummy import procedure, and the server stubs.

`TestRPC.mod`
: The client implementation of the `TestRPC` interface. It contains the import procedure and a dummy export procedure.

`TestClient.mod`
: The client implementation of the `Test` interface. It contains the client stubs.

The extensions made to Modula-2 by the researchers at DEC Systems Research Center fall into three categories: exception handling, storage allocation and deallocation, and concurrency. Each of these extensions affects the stub generation process, so they are summarized here.

The first group of extensions allows the programmer to declare, raise, and catch exceptions, using the following forms.

```
exception name(argument);

raise(name, argument);

try ... except ... end
```

The extensions having to do with storage allocation are based on a new type constructor, `ref T`, that replaces the existing `pointer to T` constructor. In addition, *automatic storage deallocation*, or *garbage collection*, is provided. The user does not explicitly deallocate objects; instead, a garbage collector deallocates objects automatically when they become unreachable from the user's program.

Standard Modula-2 has a limited form of variable-length array type, called an *open array*, that can only appear as the type of a formal parameter. Extended Modula-2 allows types of the form `ref array of T` to appear anywhere, as in the following example.

```
var a: ref array of T;
```

With this declaration, an array of any size can be created at run-time by calling `new(a,n)`; its elements are accessible as `a^[0]`,...,`a^[n-1]`.

Concurrency is available in extended Modula-2 in the form of multiple threads of control. Synchronization is provided by means of locks, condition variables, and **wait** and **signal** operations.

Exceptions and dynamic arrays must be represented in messages, so these extensions affect the externalization strategy used by the stub procedures. The presence of concurrency affects the binding procedures: the desire for concurrent threads to be able to import different instances of the same interface emphasizes the need for an explicit binding option.

The stub compiler for Modula-2 does not define a single external representation for types; instead, the external form is negotiated when a client binds to a server. The negotiation process assumes that each Modula-2 type has the same size on all machines, and that there are only two possible ways of representing integers: most significant byte first or least significant byte first. Two machines are *compatible* if they use the same representation for integers. The stub compiler can produce stubs that decide whether or not to byte-swap at run-time, based on the result of the negotiation, or it can produce more efficient stubs that never byte-swap. The results of the possible negotiations between an incompatible client and server are as follows:

1. If neither client nor server is able to byte-swap, the binding attempt fails.

2. If only one of the client/server pair is able to byte-swap, the appropriate one is told to byte-swap and the binding attempt succeeds.

3. If both client and server are able to byte-swap, the client is told to byte-swap and the binding attempt succeeds.

Normally, the stub compiler produces server stubs that cannot byte-swap and client stubs that can, with the expectation that the client can better afford the time required to byte-swap than the server. Stub compiler options allow any of the four possible combinations to be produced, however. For instance, generating server stubs with the ability to byte-swap allows the server to handle "dumb clients". An example of a dumb client might be a workstation whose bootstrap code uses client stubs to call upon network services. If the bootstrap code must fit in a limited amount of read-only memory, it might be desirable to shift the burden of byte-swapping to the server.

Parameters are passed either by value or by reference in Modula-2. A better scheme, at least for the purposes of remote procedure call, allows parameters to be declared as **in**, **out**, or **in out**. This allows the stub compiler to copy parameters in only one direction when possible.

A stub compiler must choose between in-line code or procedures to externalize data types. Using only in-line code can require enormous amounts of code space for large interfaces with complicated data types. Using only procedures can be too slow for simple data types. The Modula-2 stub compiler uses both methods, depending on the complexity of the type.

The stub compiler examines all data types that are reachable from the definition module it is processing. Reachable types are those that appear as parameters, results, and exception arguments. A marking algorithm is used to detect recursive types, which are not handled automatically. A type that is fixed-size and requires no byte-swapping is called *copyable*, because its bits can be transferred directly to and from the packet. If a type is copyable, the stub compiler externalizes it in-line.

Externalization procedures are then generated for the types that are not copyable. It is also a simple matter to declare that certain types are to be externalized by user-supplied procedures. For example, the run-time system includes such a procedure to externalize a commonly used immutable string type that cannot be handled automatically.

Pointers are normally dereferenced automatically when externalized and re-referenced when internalized. Note that sharing among pointer-containing objects is not preserved. If the programmer knows that a pointer will never be dereferenced in a remote address space, the pointer type can be declared a *handle*; in this case, only the bit pattern of the pointer is copied.

Once externalization procedures have been generated, stubs are produced for each procedure in the interface. The stub compiler first re-orders the parameter and result list so that the copyable parameters are first, and declares call and **return** packet *overlay records* for them. The call packet overlay record contains **in** parameters and the **return** packet overlay record contains **out** parameters. The **in out** parameters are present in both packets; to minimize copying in the server stub, these appear in the same position in both packets (at the beginning).

The non-copyable types are externalized by calling the appropriate procedure. The parameters to the externalization procedure include the object, a pointer to the current packet, and the current byte offset within the packet.

7.2 Observations and Lessons

Experience with the four stub compilers described above leads to a number of observations about the relationship between programming languages and remote or replicated procedure call systems.

The most important lesson is the following: *the success of a stub compiler depends on how well the interface language matches the stub language.* If there is a natural correspondence between constructs in the two languages, the resulting system is easy for programmers to use. If the correspondence is awkward or impossible, because either the interface language or the stub language lacks a particular construct, then the programmer is presented with restrictions or idioms that must be obeyed when writing client or server code. The best match, of course, occurs when the interface and stub languages are the same.

If there is a natural correspondence between an interface language and a stub language, then they are isomorphic in an informal sense. It follows that the stub languages that can be used well with a particular interface language are isomorphic both to the interface language and to each other. This observation leads the author to the conclusion that language-independent interface specification languages, like Courier [115] and Matchmaker [42], are destined to enjoy only limited success.

It was noted in Section 7.1.4 that copy-in and copy-out parameter-passing semantics are preferable to by-value and by-reference parameter-passing semantics. This lesson can be generalized: *language constructs that have only a single use are preferable to those that can disguise a multiplicity of intentions on the part of the programmer.*

The "Swiss army knife" philosophy of language design, which advocates the use of a single language construct for multiple purposes whenever possible, should be rejected in favor of "one construct, one use". If there is only a single reason to use a particular language construct, then the occurrence of that construct in a program uniquely determines the reason for its use. This makes it easier for both humans and stub compilers to understand the programmer's intention.

Here are some examples of the "one construct, one use" principle:

- Pascal and Modula-2 var parameters are used both for results and to pass large objects efficiently; copy-in/copy-out semantics are preferable, leaving the optimization to the compiler.
- Providing an explicit finalization control structure (such as unwind-protect in Lisp) is better than making the exception-handling mechanism do double duty.
- Providing both pointers and variable-length arrays in a type system is preferable to lumping them together as in C.

A final lesson concerns the limits of automation in stub compilers. There will always be data structures for which the programmer can do a better job of externalizing than the stub compiler. The designer of a stub compiler should therefore provide a convenient way for programmers to specify their own externalization procedures for particular types.

7.3 Stubs with Explicit Binding

The primary function of a stub compiler is to incorporate remote procedure calls into a programming language as transparently as possible, but sometimes it is desirable to violate this transparency. Binding is a case in point.

In a single-machine program, there is typically only one implementation module for each interface, and it is bound into the program before execution begins (if static linking is used) or when the module is referenced (if dynamic linking is used). The form of remote binding that most closely mirrors these local semantics is *implicit binding*, in which the stub compiler produces a procedure that the client calls to import a remote module. Since only one implementation of an interface can be imported at a time, the import procedure is free to keep private state information about the identity of the imported module.

Suppose a client wishes to use several servers that export the same interface: several file servers, for example. The import procedure cannot maintain global state information if the client uses the different servers concurrently. Even if the client used the different servers sequentially, the import procedure would have to be called repeatedly to change the binding.

The solution adopted in the Courier-to-C stub compiler (Section 7.1.1) and the Modula-to-Modula stub compiler (Section 7.1.4) is to surrender transparency and use *explicit binding*. When explicit binding is used, the stub compiler produces an import procedure that returns a *binding handle* to the client. This binding handle points to the state information that would otherwise have been maintained privately by the import procedure. Instead of producing stubs for the original interface, the stub compiler creates a variant interface in which each procedure is re-declared as taking an additional parameter: the binding handle. Figure 7.5 shows how explicit binding could be used by a client to perform a third-party file transfer, albeit inefficiently [69].

Using an extra parameter and a separate interface for the variant declarations has the advantage of not requiring any extensions to the language; it can be implemented solely by the stub compiler. In a language designed for distributed programming, however, an explicit

```
-- original interface to file system
filesystem:
    interface
        read: procedure (file) returns (page)
        write: procedure (file, page)
        end_of_file: procedure (file) returns (boolean)
    end interface

-- client interface produced with explicit binding option
remote_filesystem:
    interface
        binding: type = opaque
        import: procedure (location_name) returns (binding)
        read: procedure (binding, file) returns (page)
        write: procedure (binding, file, page)
        end_of_file: procedure (binding, file) returns (boolean)
    end interface

-- client of two instances of the file system interface
client:
    module
        imports remote_filesystem
        begin
            binding1 := import(server1)
            binding2 := import(server2)
            while not end_of_file(binding1, file) do
                write(binding2, file, read(binding1, file))
            end while
        end
    end module
```

Figure 7.5: Use of explicit binding

notation might be preferable. Some of the examples in this dissertation use a construction of the form `P(x) at location`. Another possibility is to allow qualified names of the form `interface.P(x)`, where `interface` is a variable rather than a constant interface name.

A binding handle can be viewed as a weak form of *capability* [27] for a remote instance of a module. In an object-based language, this approach could be used to implement remote objects.

7.4 Stubs with Explicit Replication

There are occasions when it is desirable to sacrifice replication transparency in order to take advantage of application-specific knowledge. Consider a read-only query of a replicated database [30], for example, or a broadcast protocol for resource location [11,76]. In these cases, the client only needs responses from a subset of the server troupe, and should use a collator (such as first-come) that terminates as soon as enough results are received. Another class of examples arise when a client must resolve inconsistencies among server troupe members in an application-specific manner. For instance, algorithms to maintain synchronized clocks [34,52,65], or more generally, to reach approximate byzantine agreement [24], involve averaging a set of values received from other machines. To express these algorithms in terms of replicated procedure calls, the client should use an application-specific collator to define the appropriate averaging function.

The need to define application-specific collators raises the problem of how to make replication visible at the programming language level in a consistent, expressive, and type-safe way. The solution proposed here uses an iteration construct called a *generator* (or an *iterator* in CLU) to achieve these goals [32,60,92].

A generator is a procedure that produces a sequence of results, rather than just one. In a language with generators, the conventional for loop

```
for i := 1 to 10 do
    ...
end for
```

can be replaced by

```
for i in interval(1,10) do
    ...
end for
```

where `interval` is a generator defined as follows.

```
interval:
    generator (low, high: integer) yields (i: integer)
        i := low
        while i <= high do
            yield i
            i := i + 1
        end while
    end generator
```

An activation record, or context, is created for a generator when it is first invoked at the beginning of a for loop. When a yield statement in the generator is reached, control returns to the caller along with the specified value, but unlike a conventional subroutine, the generator's context is not destroyed. When the generator is called again at the next iteration of the for loop, execution resumes in the same generator context, immediately following the yield statement. Thus, the generator and its caller obey a stylized coroutine discipline.

When the generator terminates normally (by executing its last statement), the caller's for loop terminates. If the generator terminates because of an exception, the caller's for loop terminates and the exception is raised in the enclosing context. Conversely, if control leaves the body of the for loop by means of a return, loop exit, or exception, the generator is terminated.

Some languages include constructs that specify *finalization* code to be executed whenever control leaves a particular context, typically to restore an invariant. Examples include the unwind-protect form in Lisp [97] and the finally clause in extended Modula-2 [87]. In such languages, any finalization code in an active generator that surrounds the most recently performed yield statement must be executed whenever the caller's for loop terminates.

An interesting degenerate case arises when the yield statement is used only to transfer control, without yielding any values. A generator of this type is analogous to a procedure that returns no value; it can be used solely for control flow, or it may communicate values by means of shared variables. Although this type of generator is not supported by Alphard [92], CLU [60], or Icon [32], it appears to fill a useful niche, and turns up naturally in the constructions described below. Support for it will therefore be assumed.

Generators provide an elegant means of giving the programmer control over the collating process. A stub compiler option can be used to specify that replication is to be made visible in the client or the server or both. With this explicit replication option, the stub compiler translates a procedure of the form

```
procedure (x) returns (y)
```

into generator-passing procedures of the following forms. On the client side:

```
procedure (x) returns (results: generator () yields (y))
```

On the server side:

```
procedure (arguments: generator () yields (x)) returns (y)
```

Although it would be simpler to use

```
generator (x) yields (y)
```

on the client side, the generator-returning procedure better exhibits the symmetry with the server, and simplifies the interface to the underlying replicated procedure call protocol (see Figure 7.11 below).

The degenerate form of generator (which yields no values) arises on the client side as the translation of a procedure with no results, and on the server side as the translation of a procedure with no arguments.

```
-- original interface to read-only file system
filesystem:
    interface
        read: procedure (file) returns (page)
    end interface

-- client interface produced with explicit replication option
replicated_filesystem:
    interface
        read: procedure (file) returns (results: generator () yields (page))
    end interface

-- client of file system
client:
    module
        imports replicated_filesystem
        begin
            pages := read(file)
            -- pages() generates the set of responses
            for page in pages() do
                -- exit loop when an acceptable page is received
                if acceptable(page) then exit
            end for
        end
    end module
```

Figure 7.6: A client's use of explicit replication

The violation of replication transparency entailed by the explicit replication option is reflected in the structure of the stub compiler's output. Since the types of the stub procedures do not match the types of the procedures in the original interface, variant client and server interfaces must be produced which re-declare the procedures in the above two forms. This situation is analogous to producing stubs with explicit binding handles. There are no pitfalls here for the programmer because the new type is a well-defined function of the original type and is easily used. The following two examples show this.

Figure 7.6 shows how a client can use explicit replication to short-circuit the normal process of waiting for all of the answers to a replicated call. The value returned by the client's call is a result generator, which when invoked yields each server troupe member's response. The client can stop iterating through these responses as soon as an acceptable one is found.

Figure 7.7 shows how a server can use explicit replication to collate the incoming arguments of a many-to-one call. Here the parameter to the set_temperature procedure is an argument generator: when invoked, it yields each client troupe member's argument. The set_temperature procedure computes the average of the different temperatures supplied

```
-- original interface to temperature controller
controller:
    interface
        set_temperature: procedure (temperature)
    end interface

-- server interface produced with explicit replication option
replicated_controller:
    interface
        set_temperature: procedure (arguments: generator () yields (temperature))
    end interface

-- server for temperature controller
server:
    module
        exports replicated_controller
        set_temperature:
            procedure (arguments: generator () yields (temperature))
                -- compute average value of arguments
                for temperature in arguments() do
                    sum := sum + temperature
                    n := n + 1
                end for
                average := sum/n
                -- set temperature to average
                ...
            end procedure
    end module
```

Figure 7.7: A server's use of explicit replication

```
unanimous:
    procedure (g: generator () yields (x)) returns (x)
        seen := false
        for x in g() do
            if not seen then
                representative := x
                seen := true
            else if x <> representative then
                error: disagreement
            end if
        end for
        return representative
    end procedure
```

Figure 7.8: Unanimous collator

```
first_come:
    procedure (g: generator () yields (x)) returns (x)
        for x in g() do
            -- return, terminating generator early
            return x
        end for
    end procedure
```

Figure 7.9: First-come collator

by its callers (the client troupe members) and uses this average value in its subsequent actions.

Figures 7.8, 7.9, and 7.10 demonstrate how generators can be used to program three kinds of collators. Note that the calculation of the median in Figure 7.10 can be performed in linear time [38].

It remains to be shown how the appropriate generator objects can be constructed from the underlying replicated procedure call protocol. Figure 7.11 defines a generator that yields the sequence of messages arriving from a troupe. The generator `connections(troupe)` is used to iterate over the connection records for each member of the given troupe. Each troupe is associated with a condition variable `troupe.status_change` that is signaled by the protocol implementation whenever a message arrives from any member of the troupe. The `connection.status` field indicates whether a message on that connection is still expected, has just arrived, or has already been seen. When the status of a connection is `arrived`, the `connection.message` field holds the message.

```
majority:
    procedure (g: generator () yields (x)) returns (x)
        values: array of x
        n := 0
        for x in g() do
            n := n + 1
            values[n] := x
        end for
        -- if there is a majority value it must equal the median
        m := median(values)
        -- check whether there is a majority value by counting
        count := 0
        for i := 1 to n do
            if values[i] = m then
                count := count + 1
            end if
        end for
        if count > n/2 then
            return m
        else
            error: no majority
        end if
    end procedure
```

Figure 7.10: Majority collator

```
messages:
    generator (troupe) yields (message)
        loop
            in_progress := false
            new_arrival := false
            for conn in connections(troupe) do
                case conn.status of
                    expected:
                        in_progress := true
                        exit
                    arrived:
                        new_arrival := true
                        connection := conn
                        exit
                    seen:
                        -- continue iterating
                end case
            end for
            if new_arrival then
                yield connection.message
            else if in_progress then
                await troupe.status_change
            else
                -- all messages have been seen
                exit
            end if
        end loop
    end generator
```

Figure 7.11: Generator of messages from a troupe

7.5 Programming-in-the-Large

A configuration is a mapping from troupes to sets of machines. A configuration language allows the programmer to specify the set of acceptable configurations of a replicated distributed program. A configuration manager uses this specification to perform the necessary creation and binding of troupe members when the program is started or dynamically reconfigured.

A configuration language is an important programming-in-the-large tool. Without such a tool, programmers would be forced to resort to *ad hoc* methods of instantiating and reconfiguring distributed programs.

7.5.1 Configuration Languages

The idea of a *module interconnection language* was introduced by DeRemer and Kron [21] after the initial work on the subject by Parnas [77]. Practical examples of such languages include the Mesa configuration language C/Mesa [68] and the Cedar system modeling language SML [58,89].

In a distributed environment, a *distributed configuration language* is required [73]. In addition to providing the functions of a module interconnection language, a distributed configuration language allows the programmer to control the mapping of modules to machines. Replicated distributed programs require still more functionality, namely the ability to specify replication on a module-by-module basis. It should be possible to specify the degree of replication of individual modules without having to modify them. In order to handle dynamic reconfiguration, this service must be available to the program while it is executing.

7.5.2 A Troupe Configuration Language

The troupe configuration language introduced in this section allows a programmer to specify the configuration of a troupe (the degree of replication of the troupe and on which machines the troupe members should execute) without modifying the source code for the module being replicated.

Mappings between troupes and sets of machines are specified in terms of the required *attributes* of the machines. A similar approach was taken in the design of the Resource Manager for the Cambridge Ring [20,72]. This approach allows the programmer to specify those machine attributes that are important for each troupe member, while leaving unspecified those that are irrelevant. If more than one machine has the required attributes, so much the better: there will be more latitude in instantiating and reconfiguring the troupe, and more opportunity for optimization.

The troupe configuration language is an extension of propositional logic with variables that range over the machines in the distributed system. Each machine possesses an extensible list of attributes, which are simply pairs of names and values. Values may be strings, numbers, or truth values. For example, a machine m might have the attributes

$$\langle(\text{name}, \text{"UCB-Monet"}), (\text{memory}, 10), (\text{has-floating-point}, \textbf{true})\rangle$$

to indicate that its name is "UCB-Monet", it has 10 megabytes of memory, and it has floating point hardware. Note that the name of a machine is just another attribute.

$$\begin{array}{lcl}
\langle\textbf{formula}\rangle & : & (\langle\textbf{formula}\rangle) \\
 & | & \langle\textbf{formula}\rangle \textbf{ and } \langle\textbf{formula}\rangle \\
 & | & \langle\textbf{formula}\rangle \textbf{ or } \langle\textbf{formula}\rangle \\
 & | & \textbf{not } \langle\textbf{formula}\rangle \\
 & | & \langle\textbf{variable}\rangle.\langle\textbf{property}\rangle \\
 & | & \langle\textbf{term}\rangle\ \langle\textbf{relation}\rangle\ \langle\textbf{term}\rangle \\
\langle\textbf{term}\rangle & : & \langle\textbf{variable}\rangle.\langle\textbf{attribute}\rangle \\
 & | & \langle\textbf{constant}\rangle \\
\langle\textbf{relation}\rangle & : & = \mid \neq \mid < \mid \leq \mid > \mid \geq \\
\langle\textbf{constant}\rangle & : & \langle\textbf{string}\rangle \mid \langle\textbf{number}\rangle
\end{array}$$

Figure 7.12: Configuration language for troupes

The following is an example of a formula $\varphi(x)$ in the troupe configuration language.

$$x.\text{name} = \text{“UCB-Monet”} \textbf{ and } x.\text{memory} = 10 \textbf{ and } x.\text{has-floating-point}$$

The machine m of the previous example is said to *satisfy* the formula $\varphi(x)$ because $\varphi(m)$ is true. A Boolean-valued attribute such as "has-floating-point" is called a *property*; distinguishing properties from other attributes makes the Boolean constants **true** and **false** unnecessary. The BNF grammar for this language is shown in Figure 7.12.

A formula $\varphi(x_1, \ldots, x_n)$ in the language of Figure 7.12 is used to specify a troupe as follows:

$$\textbf{troupe}(x_1, \ldots, x_n) \textbf{ where } \varphi(x_1, \ldots, x_n)$$

The troupe members are required to be distinct; the language contains no provision for comparing two machines for equality, only for comparing attribute values. Since any troupe that satisfies the specification $\varphi(x_1, \ldots, x_n)$ must have n members, it is impossible to specify a troupe of variable size. It remains to be seen whether this restriction will prove inconvenient in practice.

7.5.3 A Troupe Configuration Manager

The troupe configuration manager described here uses a set of troupe specifications to instantiate a troupe and to reconfigure it. Reconfiguration can occur either after a partial failure, as discussed in Section 6.4, or because a troupe specification has been changed. Both instantiating a troupe and reconfiguring a troupe are instances of the following *troupe extension* problem: given a troupe specification $\varphi(x_1, \ldots, x_n)$, a universe U of machines and their attributes, and a particular set of machines $M \subseteq U$, find a new set $M' = \{m_1, \ldots, m_n\} \subseteq U$ *that satisfies* φ *and is as close to* M *as possible*. The latter condition can be formalized as follows. Let $S \ominus T$ denote the symmetric set difference $(S - T) \cup (T - S)$ and let $|S|$ denote the cardinality of set S. Then M' satisfies φ and is as close to M as possible if

$$\varphi(M') \land \forall S\, [\varphi(S) \Rightarrow |S \ominus M| \geq |M' \ominus M|]$$

Note that the instantiation problem is just the case where $M = \emptyset$.

A configuration manager for troupes can therefore be based on a single procedure

```
extend_troupe:
    procedure (specification, old_troupe) returns (new_troupe)
```

that searches a database of machine attributes for acceptable troupe extensions. A version of this procedure that handles only instantiation was implemented in Lisp [28], using backtracking to perform an exhaustive search. The exponential-time complexity of this procedure is acceptable given the small number of variables (troupe members) in most troupe specifications.

A full configuration manager requires server processes at each machine to handle the details of module instantiation. Under Berkeley 4.2BSD [43], for example, the system utilities for remote file transfer and remote command execution can be used for this purpose.

Chapter 8

Conclusion

This chapter summarizes the contributions of this dissertation and suggests some areas for further research.

8.1 Summary

A new software architecture for fault-tolerant distributed programs is presented. The mechanisms described in this dissertation allow a programmer to add replication transparently and flexibly to existing programs. The resulting replicated distributed programs automatically tolerate partial failures of the underlying fail-stop hardware.

The architecture combines remote procedure calls with replication of program modules for fault tolerance. The replicated modules, called troupes, are the basis for constructing replicated distributed programs.

Previous fault-tolerant architectures were either too expensive or too inflexible. Simple replication of hardware components, for example, requires all software to be executed redundantly, rather than just critical modules, and permits only a single degree of replication. In contrast, the present approach introduces replication at the program module level, and allows the degree of replication of each module to vary independently and dynamically.

The replication mechanisms introduced in this dissertation can be used transparently, so that the details of replication are invisible to the programmer. Transparent distributed and replicated mechanisms are an important means of coping with the complexity of fault-tolerant distributed programs. A formal model of program semantics is used to characterize deterministic programs, a class of programs that can be transparently replicated.

The model is based on program modules and threads of control. In a distributed system, threads must be able to cross machine boundaries to move between modules on different machines. An algorithm to simulate such distributed threads in terms of conventional processes and remote procedure calls is presented.

Transfer of control between troupes requires generalizing remote procedure calls to replicated procedure calls. The semantics of replicated procedure calls can be summarized as exactly-once execution at all replicas.

The Circus replicated procedure call implementation is described. Message transport is provided by a datagram-based paired message layer. The general replicated procedure call protocol, requiring many-to-many communication, is expressed in terms of two subprotocols, for the one-to-many and many-to-one cases.

In the Circus system, each troupe member waits for all incoming messages before proceeding. Troupe members are thus synchronized at each replicated procedure call and return. Alternative schemes that allow computation to proceed before all messages have arrived are discussed.

Experiments were conducted to measure the performance of the Circus replicated procedure call implementation. The results of the measurements show that six Berkeley 4.2BSD system calls account for more than half of the CPU time of a Circus replicated procedure call. The two most expensive of these system calls use a particularly inefficient interface to copy data between user and kernel address spaces. The other four system calls are used to compensate for the lack of lightweight processes in Berkeley 4.2BSD.

The use of transactions for synchronizing concurrent threads of control within replicated distributed programs is discussed. Serializability, the property guaranteed by concurrency control algorithms for conventional transactions, is shown to be insufficient for the purposes of replicated transactions, because it does not guarantee that transactions commit in the same order at all troupe members.

A troupe commit protocol that guarantees a consistent commit order for replicated transactions is presented, and a probabilistic model is used to analyze its performance. The analysis shows that the protocol might not make any progress if there are many conflicting transactions. An alternative approach, based on an ordered broadcast protocol, is presented.

Mechanisms for binding and reconfiguring replicated distributed programs are described. The problem of detecting obsolete binding information is identified; this problem is both more complicated and more critical than the corresponding problem in the unreplicated case. A solution using troupe IDs as incarnation numbers is presented.

A probabilistic model of troupe reliability is used to analyze when to reconfigure a troupe after a partial failure. The results of the analysis relate the lifetime, replacement time, and degree of replication of troupe members to the overall availability of the troupe.

Issues relating to programming languages and environments for reliable applications are discussed. At the programming-in-the-small level, stub compilers are used to integrate remote and replicated procedure calls into programming languages. Four stub compilers are described, and a number of lessons and observations are presented.

The replication mechanisms introduced in this dissertation can be used explicitly, so that the details of replication are accessible to the programmer. The stub compiler approach is used to give the programmer explicit access to replication in a powerful and type-safe way.

It is important for programmers to be able to specify and control the configuration of replicated distributed programs at the programming-in-the-large level. Two tools were designed for this purpose: a troupe configuration language, for specifying acceptable mappings of troupe members to machines; and a troupe configuration manager, for instantiating and reconfiguring troupes.

8.2 Directions for Future Research

Replicated procedure calls are useful for more than just fully replicated distributed programs. The two-phase commit and ordered broadcast protocols in Chapter 5 are examples of how the use of replicated procedure calls leads to an elegant formulation of algorithms traditionally described in terms of asynchronous messages.

An important area for further research is to express more algorithms of this type in terms of replicated procedure calls. For example, the algorithms used in distributed database systems for concurrency control, replicated data, atomic commit and recovery, and deadlock detection would lend themselves to such treatment.

Further research is needed to evaluate the alternative replicated procedure call protocols described in Section 4.3.4, and to discover new ones. An approach that allowed the choice between such schemes to be made on a per-module basis, as a programming-in-the-large activity, would be attractive.

The replicated transaction algorithms presented in Sections 5.3 and 5.4 must be implemented and their performance evaluated. Allowing application-specific concurrency control within the context of troupes is another area for further work.

A fully functional configuration language and manager should be implemented, along the lines discussed in Section 7.5.3. A graphical user interface would provide a powerful means of manipulating module interconnections.

Recall that generators were adopted in Section 7.4 to permit explicit use of replication at the programming language level. Programming languages should support generators as first-class objects, which includes allowing generators to appear in interfaces. Generators should also be callable remotely to preserve transparency.

Nelson describes a general mechanism for remote transfer of control between contexts that suffices to implement coroutines and other control structures [73]. Remote transfers can be implemented directly at the protocol level using reliable messages, or they can be translated by the stub compiler into remote procedure calls.

Since generators embody a restricted form of coroutine discipline, remote generator invocations do not require the full generality of the remote transfer operation. It may be possible for a transport level protocol to use this restriction to advantage, much as the paired message protocol uses the call/return semantics of procedure calls to reduce the number of acknowledgments. Designing a single protocol that handles procedures, generators, and exceptions efficiently is a challenge for future researchers.

References

[1] J. E. Allchin and M. S. McKendry. Synchronization and recovery of actions. *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, August 1983, pages 31–44. *Operating Systems Review* 19(1), January 1985, pages 32–45.

[2] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice.* Prentice-Hall, 1981.

[3] Joel F. Bartlett. A NonStop kernel. *Proceedings of the 8th Symposium on Operating Systems Principles.* *Operating Systems Review* 15(5), December 1981, pages 22–29.

[4] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *Computing Surveys* 13(2), June 1981, pages 185–221.

[5] Kenneth P. Birman, Thomas A. Joseph, and Thomas Räuchle. Concurrency Control in Resilient Objects. Report TR 84-622, Department of Computer Science, Cornell University, July 1984.

[6] Kenneth P. Birman, Amr El Abbadi, Wally Dietrich, Thomas A. Joseph, and Thomas Räuchle. An Overview of the ISIS Project. Report TR 84-642, Department of Computer Science, Cornell University, October 1984.

[7] Kenneth P. Birman and Thomas A. Joseph. Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems. Report TR 84-644, Department of Computer Science, Cornell University, October 1984.

[8] Kenneth P. Birman, Thomas A. Joseph, Thomas Räuchle, and Amr El Abbadi. Implementing fault-tolerant distributed objects. *Proceedings of the 4th Symposium on Reliability in Distributed Software and Database Systems*, October 1984, pages 124–133.

[9] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM* 25(4), April 1982, pages 260–274.

[10] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems* 2(1), February 1984, pages 39–59.

[11] David Reeves Boggs. *Internet Broadcasting.* Ph.D. dissertation, Department of Electrical Engineering, Stanford University, October 1983. Report CSL-83-3, Xerox PARC.

[12] Anita Borg, Jim Baumbach, and Sam Glazer. A message system supporting fault tolerance. *Proceedings of the 9th ACM Symposium on Operating Systems Principles.* *Operating Systems Review* 17(5), October 1983, pages 90–99.

[13] Liming Chen and Algirdas Avizienis. *N*-version programming: A fault-tolerance approach to reliability of software operation. *Digest of Papers, FTCS-8: 8th Annual International Conference on Fault-Tolerant Computing*, June 1978, pages 3–9.

[14] David R. Cheriton and Willy Zwaenepoel. One-to-Many Interprocess Communication in the V-System. Report STAN-CS-84-1011, Department of Computer Science, Stanford University, August 1984.

[15] J. G. Cleary. Process handling on Burroughs B6500. *Proceedings of the 4th Australian Computer Conference*, Adelaide, South Australia, 1969, pages 231–239.

[16] Melvin E. Conway. A multiprocessor system design. *Proceedings of the AFIPS 1963 Fall Joint Computer Conference*, volume 24, pages 139–146.

[17] Eric C. Cooper.
Analysis of distributed commit protocols.
Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data, June 1982, pages 175–183.

[18] Eric C. Cooper.
Replicated procedure call.
Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing, August 1984, pages 220–232.

[19] Eric C. Cooper.
Circus: A replicated procedure call facility.
Proceedings of the 4th Symposium on Reliability in Distributed Software and Database Systems, October 1984, pages 11–24.

[20] Daniel H. Craft.
Resource management in a decentralized system.
Proceedings of the 9th ACM Symposium on Operating Systems Principles.
Operating Systems Review 17(5), October 1983, pages 11–19.

[21] Frank DeRemer and Hans Kron.
Programming-in-the-large versus programming-in-the-small.
Proceedings of the 1975 International Conference on Reliable Software, April 1975, pages 114–121.

[22] Digital Equipment Corporation, Intel Corporation, and Xerox Corporation.
The Ethernet: A Local Area Network.
September 1980.

[23] E. W. Dijkstra.
Cooperating sequential processes.
In *Programming Languages*, edited by F. Genuys. Academic Press, 1968, pages 43–112.

[24] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl.
Reaching approximate agreement in the presence of faults.
Proceedings of the 3rd Symposium on Reliability in Distributed Software and Database Systems, October 1983, pages 145–154.

[25] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger.
The notions of consistency and predicate locks in a database system.
Communications of the ACM 19(11), November 1976, pages 624–633.

[26] R. S. Fabry.
Dynamic verification of operating system decisions.
Communications of the ACM 16(11), November 1973, pages 659–668.

[27] R. S. Fabry.
Capability-based addressing.
Communications of the ACM 17(7), July 1974, pages 403–412.

[28] John K. Foderaro, Keith L. Sklower, and Kevin Layer.
The Franz Lisp Manual.
Computer Science Division, University of California, Berkeley, June 1983.

[29] D. P. Friedman and D. S. Wise.
CONS should not evaluate its arguments.
In *Automata, Languages, and Programming*, edited by S. Michaelson and R. Milner. Edinburgh University Press, 1976, pages 257–284.

[30] David K. Gifford.
Weighted voting for replicated data.
Proceedings of the 7th Symposium on Operating Systems Principles.
Operating Systems Review 13(5), December 1979, pages 150–162.

[31] J. N. Gray.
Notes on data base operating systems.
In *Operating Systems: An Advanced Course*, edited by R. Bayer, R. M. Graham, and G. Seegmüller. Lecture Notes in Computer Science, volume 60, Springer-Verlag, 1978, pages 393–481.

[32] Ralph E. Griswold, David R. Hanson, and John T. Korb.
Generators in Icon.
ACM Transactions on Programming Languages and Systems 3(2), April 1981, pages 144–161.

[33] Per Gunningberg.
Voting and redundancy management implemented by protocols in distributed systems.
Digest of Papers, FTCS-13: 13th International Symposium on Fault-Tolerant Computing, June 1983, pages 182–185.

[34] Joseph Y. Halpern, Barbara Simons, and Ray Strong.
Fault-tolerant clock synchronisation.
Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing, August 1984, pages 89–102.

[35] P. Henderson and J. H. Morris, Jr.
A lazy evaluator.
Conference Record of the 3rd ACM Symposium on Principles of Programming Languages, January 1976, pages 95–103.

[36] M. Herlihy and B. Liskov.
A value transmission method for abstract data types.
ACM Transactions on Programming Languages and Systems 4(4), October 1982, pages 527–551.

[37] Maurice Peter Herlihy.
Replication Methods for Abstract Data Types.
Ph.D. dissertation, Department of Electrical Engineering and Computer Science, MIT, May 1984.
Report MIT/LCS/TR-319.

[38] C. A. R. Hoare.
Algorithms 63–65: Partition, Quicksort, and Find.
Communications of the ACM 4(7), July 1961, pages 321–322.

[39] C. A. R. Hoare.
Monitors: An operating system structuring concept.
Communications of the ACM 17(10), October 1974, pages 549–557.

[40] Warren H. Jessop, Jerre D. Noe, David M. Jacobson, Jean-Loup Baer, and Calton Pu.
The Eden transaction-based file system.
Proceedings of the 2nd Symposium on Reliability in Distributed Software and Database Systems, July 1982, pages 163–169.

[41] Stephen C. Johnson.
Yacc: Yet Another Compiler-Compiler.
Computing Science Technical Report 32, Bell Laboratories, July 1975.

[42] Michael B. Jones, Richard F. Rashid, and Mary R. Thompson.
Matchmaker: An interface specification language for distributed processing.
Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages, January 1985, pages 225–235.

[43] William Joy, Eric Cooper, Robert Fabry, Samuel Leffler, Kirk McKusick, and David Mosher.
4.2BSD System Manual.
Computer Systems Research Group, Computer Science Division, University of California, Berkeley, July 1983,

[44] Brian W. Kernighan and Dennis M. Ritchie.
The C Programming Language.
Prentice-Hall, 1978.

[45] Leonard Kleinrock.
Queueing Systems, Volume 1: Theory.
John Wiley and Sons, 1975.

[46] Donald E. Knuth.
The Art of Computer Programming, Volume 1: Fundamental Algorithms.
2nd edition, Addison-Wesley, 1973.

[47] Walter H. Kohler.
A survey of techniques for synchronization and recovery in decentralized computer systems.
Computing Surveys 13(2), June 1981, pages 149–183.

[48] H. T. Kung and John T. Robinson.
On optimistic methods for concurrency control.
ACM Transactions on Database Systems 6(2), June 1981, pages 213–226.

[49] Leslie Lamport.
The implementation of reliable distributed multiprocess systems.
Computer Networks 2(2), May 1978, pages 95–114.

[50] Leslie Lamport.
Time, clocks, and the ordering of events in a distributed system.
Communications of the ACM 21(7), July 1978, pages 558–565.

[51] Leslie Lamport, Robert Shostak, and Marshall Pease.
The byzantine generals problem.
ACM Transactions on Programming Languages and Systems 4(3), July 1982, pages 382–401.

[52] Leslie Lamport and P. M. Melliar-Smith.
Byzantine clock synchronisation.
Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing, August 1984, pages 68–74.

[53] Butler W. Lampson and Howard E. Sturgis.
Crash Recovery in a Distributed Data Storage System.
Unpublished paper, Computer Science Laboratory, Xerox PARC.

[54] Butler W. Lampson and David D. Redell.
Experience with processes and monitors in Mesa.
Communications of the ACM 23(2), February 1980, pages 105–117.

[55] Butler W. Lampson.
Replicated Commit.
Unpublished paper, Computer Science Laboratory, Xerox PARC, January 1981.

[56] B. W. Lampson, M. Paul, and H. J. Siegert, editors.
Distributed Systems—Architecture and Implementation: An Advanced Course.
Lecture Notes in Computer Science, volume 105, Springer-Verlag, 1981.

[57] Butler W. Lampson.
Atomic transactions.
In *Distributed Systems* [56], pages 246–265.

[58] Butler W. Lampson and Eric E. Schmidt.
Practical use of a polymorphic applicative language.
Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages, January 1983, pages 237–255.

[59] Gerard Le Lann.
Synchronisation.
In *Distributed Systems* [56], pages 266–283.

[60] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert.
Abstraction mechanisms in CLU.
Communications of the ACM 20(8), August 1977, pages 564–576.

[61] Barbara Liskov and Robert Scheifler.
Guardians and actions: Linguistic support for robust, distributed programs.
ACM Transactions on Programming Languages and Systems 5(3), July 1983, pages 381–404.

[62] Barbara Liskov, Maurice Herlihy, Paul Johnson, Gary Leavens, Robert Scheifler, and William Weihl.
Preliminary Argus Reference Manual.
Programming Methodology Group Memo 39, MIT Laboratory for Computer Science, October 1983.

[63] Barbara Liskov.
Overview of the Argus Language and System.
Programming Methodology Group Memo 40, MIT Laboratory for Computer Science, February 1984.

[64] D. B. Lomet.
Process structuring, synchronisation, and recovery using atomic actions.
Proceedings of the ACM Conference on Language Design for Reliable Software.
SIGPLAN Notices 12(3), March 1977, pages 128–137.

[65] Jennifer Lundelius and Nancy Lynch.
A new fault-tolerant algorithm for clock synchronisation.
Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing, August 1984, pages 75–88.

[66] R. E. Lyons and W. Vanderkulk.
The use of triple-modular redundancy to improve computer reliability.
IBM Journal of Research and Development 6(2), April 1962, pages 200–209.

[67] Robert M. Metcalfe and David R. Boggs.
Ethernet: Distributed packet switching for local computer networks.
Communications of the ACM 19(7), July 1976, pages 395–403.

[68] James G. Mitchell, William Maybury, and Richard Sweet.
Mesa Language Manual, Version 5.0.
Report CSL-79-3, Xerox PARC, April 1979.

[69] Jeffrey Mogul.
Private communication, February 1983.

[70] J. Eliot B. Moss.
Nested Transactions: An Approach to Reliable Distributed Computing.
Ph.D. dissertation, Department of Electrical Engineering and Computer Science, MIT, April 1981.
Report MIT/LCS/TR-260.

[71] J. Eliot B. Moss.
Nested transactions and reliable distributed computing.
Proceedings of the 2nd Symposium on Reliability in Distributed Software and Database Systems, July 1982, pages 33–39.

[72] R. M. Needham and A. J. Herbert.
The Cambridge Distributed Computing System.
Addison-Wesley, 1982, pages 114–123.

[73] Bruce Jay Nelson.
Remote Procedure Call.
Ph.D. dissertation, Computer Science Department, Carnegie-Mellon University, May 1981.
CMU report CMU-CS-81-119 and Xerox PARC report CSL-81-9.

[74] Bruce Nelson and Andrew Birrell.
Lupine User's Guide: An Introduction to Remote Procedure Calls in Cedar.
The Cedar Manual, Computer Science Laboratory, Xerox PARC, July 1982.

[75] Ron Obermarck.
Distributed deadlock detection algorithm.
ACM Transactions on Database Systems 7(2), June 1982, pages 187–208.

[76] Derek C. Oppen and Yogen K. Dalal.
The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment.
Report OPD-T8103, Xerox Office Products Division, October 1981.

[77] D. L. Parnas.
A technique for software module specification with examples.
Communications of the ACM 15(5), May 1972, pages 330–336.

[78] M. Pease, R. Shostak, and L. Lamport.
Reaching agreement in the presence of faults.
Journal of the ACM 27(2), April 1980, pages 228–234.

[79] W. H. Pierce.
Adaptive vote-takers improve the use of redundancy.
In *Redundancy Techniques for Computing Systems*, edited by Richard H. Wilcox and William C. Mann. Spartan Books, Washington, D.C., 1962, pages 229–250.

[80] Jon Postel.
User Datagram Protocol.
RFC 768, Information Sciences Institute, University of Southern California, August 1980.

[81] Jon Postel.
Internet Protocol.
RFC 791, Information Sciences Institute, University of Southern California, September 1981.

[82] Jon Postel.
Transmission Control Protocol.
RFC 793, Information Sciences Institute, University of Southern California, September 1981.

[83] Michael L. Powell.
A portable optimizing compiler for Modula-2.
Proceedings of the SIGPLAN '84 Symposium on Compiler Construction.
SIGPLAN Notices 19(6), June 1984, pages 310–318.

[84] David P. Reed.
Naming and Synchronization in a Decentralized Computer System.
Ph.D. dissertation, Department of Electrical Engineering and Computer Science, MIT, September 1978.
Report *MIT/LCS/TR-205.*

[85] David P. Reed.
Implementing atomic actions on decentralized data.
ACM Transactions on Computer Systems 1(1), February 1983, pages 3–23.

[86] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis.
System level concurrency control for distributed database systems.
ACM Transactions on Database Systems 3(2), June 1978, pages 178–198.

[87] Paul Rovner, Roy Levin, and John Wick.
On Extending Modula-2 for Building Large, Integrated Systems.
DEC Systems Research Center Report 3, January 1985.

[88] Richard D. Schlichting and Fred B. Schneider.
Fail-stop processors: An approach to designing fault-tolerant computing systems.
ACM Transactions on Computer Systems 1(3), August 1983, pages 222–238.

[89] Eric Emerson Schmidt.
Controlling Large Software Development in a Distributed Environment.
Ph.D. dissertation, Computer Science Division, University of California, Berkeley, December 1982.
Report CSL-82-7, Xerox PARC.

[90] Fred B. Schneider.
Byzantine generals in action: Implementing fail-stop processors.
ACM Transactions on Computer Systems 2(2), May 1984, pages 145–154.

[91] Peter M. Schwarz and Alfred Z. Spector.
Synchronizing shared abstract data types.
ACM Transactions on Computer Systems 2(3), August 1984, pages 223–250.

[92] Mary Shaw, William A. Wulf, and Ralph L. London.
Abstraction and verification in Alphard: Defining and specifying iteration and generators.
Communications of the ACM 20(8), August 1977, pages 553–564.

[93] John F. Shoch and Jon A. Hupp.
The "Worm" programs: Early experience with a distributed computation.
Communications of the ACM 25(3), March 1982, pages 172–180.

[94] Jonathan Sieber.
TRIX: A Communications Oriented Operating System.
M.S. thesis, Department of Electrical Engineering and Computer Science, MIT, 1983.

[95] Dale Skeen.
Atomic broadcasts.
In preparation. Birman and Joseph [7] briefly describe one of Skeen's algorithms in an appendix.

[96] Alfred Z. Spector and Peter M. Schwarz.
Transactions: A construct for reliable distributed computing.
Operating Systems Review 17(2), April 1983, pages 18–35.

[97] Guy L. Steele, Jr.
Common Lisp: The Language.
Digital Press, 1984.

[98] H. R. Strong and D. Dolev.
Byzantine agreement.
Digest of Papers, Spring COMPCON 83: 26th IEEE Computer Society International Conference, February 1983, pages 77–81.

[99] H. Sturgis, J. Mitchell, and J. Israel.
Issues in the design and use of a distributed file system.
Operating Systems Review 14(3), July 1980, pages 55–69.

[100] Sun Microsystems.
Remote Procedure Call Reference Manual.
Mountain View, California, October 1984.

[101] Liba Svobodova.
A reliable object-oriented data repository for a distributed computer system.
Proceedings of the 8th Symposium on Operating Systems Principles.
Operating Systems Review 15(5), December 1981, pages 47–58.

[102] Tandem Computers.
GUARDIAN Operating System Programming Manual.
Cupertino, California, 1982.

[103] Warren Teitelman.
Interlisp Reference Manual.
Xerox PARC, 1974.

[104] Irving L. Traiger, Jim Gray, Cesare A. Galtieri, and Bruce G. Lindsay.
Transactions and consistency in distributed database systems.
ACM Transactions on Database Systems 7(3), September 1982, pages 323–342.

[105] United States Department of Defense.
Reference Manual for the Ada Programming Language.
ANSI/MIL-STD-1815A-1983, U.S. Government Printing Office, February 1983.

[106] J. von Neumann.
Probabilistic logics and the synthesis of reliable organisms from unreliable components.
In *Automata Studies*, edited by C. E. Shannon and J. McCarthy. Princeton University Press, 1956, pages 43–98.

[107] William Weihl and Barbara Liskov.
Specification and implementation of resilient, atomic data types.
Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems.
SIGPLAN Notices 18(6), June 1983, pages 53–64.

[108] William E. Weihl.
Data-dependent concurrency control and recovery.
Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing, August 1983, pages 73–74.
Operating Systems Review 19(1), January 1985, pages 19–31.

[109] William E. Weihl.
Specification and Implementation of Atomic Data Types.
Ph.D. dissertation, Department of Electrical Engineering and Computer Science, MIT, March 1984.
Report MIT/LCS/TR-314.

[110] John H. Wensley.
SIFT—Software implemented fault tolerance.
Proceedings of the AFIPS 1972 Fall Joint Computer Conference, volume 41, part I, December 1972, pages 243–253.

[111] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P. M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock.
SIFT: Design and analysis of a fault-tolerant computer for aircraft control.
Proceedings of the IEEE 66(10), October 1978, pages 1240–1255.

[112] Karen White.
Implementation of Remote Procedure Call in the Berkeley UNIX Kernel.
M.S. report, Computer Science Division, University of California, Berkeley. In preparation.

[113] Niklaus Wirth.
Programming in Modula-2.
2nd edition, Springer-Verlag, 1983.

[114] Xerox Corporation.
Internet Transport Protocols.
Xerox System Integration Standard 028112, December 1981.

[115] Xerox Corporation.
Courier: The Remote Procedure Call Protocol.
Xerox System Integration Standard 038112, December 1981.

[116] Gary York, Daniel Siewiorek, and Zary Segall.
Asynchronous software voting in NMR computer structures.
Proceedings of the 3rd Symposium on Reliability in Distributed Software and Database Systems, October 1983, pages 28–37.